UNIVERSITY OF OSLO
Department of Informatics

Evaluation of Wireless Sensor Networks for Environmental Monitoring

Jan Egil Vestbø
janeves@ifi.uio.no

Spring 2013

Abstract

Wireless Sensor Network (WSN) is an evolving technology enabled by improvements in technology. A WSN is a wireless network of nodes that are small in size, and poses challenges like limited power supply and processing power, and lossy radio links. These networks are often used to collect data from sensors at multiple locations. In order to increase lifetime these devices must minimize power consumption. The lossy links between poses challenges that normal computer network routing protocols does not solve very well. IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) is a routing protocol designed specifically for WSN, and is made for communication flows that are normal in WSNs. RPL uses an Objective Function (OF) in order to optimize routes, which enables different optimization objectives.

In this thesis a WSN using RPL is implemented and deployed. RPL specifies the use of an OF, which uses metrics in order to fulfill its optimization goals. This thesis implements RPL in TinyOS, an operating system designed specifically for embedded systems like WSNs. This implementation proposes to use LQI, an link quality indicator for 802.15.4 radios which are often used in WSNs, as an initial value to the Estimated Transmission Count (ETX). The initial is an improvement on other implementation using ETX in that it passively estimates an ETX for all nodes. Nodes thus have an indication as to how good the link towards nodes that have not been tested is. This estimation can also be kept up to date for possible parents that are not actively tested. My test shows that proper calculation from LQI to ETX is very important for this to provide good results.

The resulting implementation is used in a deployed network consisting of 13 nodes that measure temperature and humidity in the church in Laksevåg. The tested values for the LQI to ETX conversion was made very conservative in order to minimize loss. This resulted in routes where the hop count was very high, compared to routes that were selected with a different calculation, which showed shorter routes in terms of hop, while still providing a high delivery ratio. The LQI to ETX values need to be set correctly in order to provide the wanted routes. This is left for future work.

Contents

List of Figures

Preface

This thesis completes my Master Degree at the Department of Informatics at University of Oslo.

I would like to thank my supervisor Knut Øvsthus at Bergen University College for providing excellent support, guidance and motivation. And also providing an interesting thesis.

I wish to thank Anders Taranger for previous done work and help regarding his code. I would also like to thank Paul Ottar Tornes and the rest of the people at Akasia and Bergen Kirkelige Fellesråd for providing access to the church in Laksevåg, an environment with a need for a Wireless Sensor Network. Thanks to Bergen University College for providing office space, and my friend and colleague Joachim Tingvold for company and support. I would also thank my family and friends for love and support.

Chapter 1

Introduction

Advancements in technology have enabled the use of technology in new areas. Wireless Sensor Networks consists of small devices. These devices are often spread out, and enacts in a wireless self-organized network. These networks often gather information from their surroundings (e.g., temperature, humidity), which are relayed through the network. Because of their size, these devices operate with limited resources, they are often battery operated, have low memory and processing resources, and utilizes lossy radio links. Many normal computer and computer network solutions do not have these constraints, and therefore does not translate very well to the requirements and limitations of sensor networks. One example of this is network routing protocols. New solutions are therefore created with these limitations in mind. One example of this is the dynamic routing protocol IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL). This thesis will focus on data-collection WSNs. These are networks where nodes report information to a common point, a base station.

1.1 Overview

This thesis will introduce WSN concepts and technology in chapter 2. The RPL routing protocol will thereafter be explained in chapter 3. The IRIS that will be used to implement and deploy a WSN with RPL is then described in chapter 4. An Objective Function (OF) is chosen in chapter 5. The OF is used by RPL to optimize routes based on given criteria. This chapter also presents measurements on links using the IRIS nodes, which is used to select the optimization results used by the OF. TinyOS is an operating system specifically made for low powered devices like the ones used in WSN. TinyOS will be used to implement a WSN with RPL and data measuring capabilities. Chapter 6 introduces TinyOS, and gives an overview over a TinyOS application. TinyOS components that have been used are also described. The resulting implementation of RPL, a data-forwarding layer, and a measuring application will be described in chapter 7. Chapter 8 thereafter describes the deployment of the WSN. Results and experiences from this deployment are also discussed in this chapter.

Chapter 2

Wireless Sensor Networks

This chapter introduces Wireless Sensor Networks (WSN). Some applications and a general introduction is given in section 2.1. The basic building bricks, sensor nodes, are thereafter explained in section 2.2. Base stations, used to connect WSNs to other networks or computers, are described in section 2.3. Different models on data reporting used by WSNs are then described in section 2.3. Section 2.5 lastly describes different topologies and data flows often used in WSNs.

2.1 Introduction

The advancements in technology have enabled us to make devices that are inexpensive, small, and require very little power to operate. New applications are made possible through this technology. Wireless Sensor Network(WSN) is one of the technologies that has been realized because of this. Alot of research has been, and is still being done,in the area of Wireless Sensor Networks. WSNs consist of sensor nodes, these sensor nodes are low cost devices that often are small in size. They have the ability to sense the environment, process these readings and communicate this information to wirelessly other nodes. These readings might include temperature, light intensity, vibrations, pressure and movement.

The small node size makes it possible for nodes to be discreet in its environment. Combined with the low sensor node price, it is feasible to deploy a high density of nodes in an environment (e.g., up to several dozen/m^3). This makes high-resolution sensing possible. The information is gathered and centralized by the use of wireless communication between the sensor nodes.

Example uses of WSNs:

- *Forest fire detection* – Detects and alarms when a forest fire has started.
- *Toxic detection* – May be used by military for threats, or by civilian in areas in danger of toxic spills.
- *Control of light and temperature* – Used in indoor environment to control light, and temperature conditions based on the location of people.
- *Building Monitoring* – Used to monitor buildings and their structural movement.

The areas where a sensor network is to be deployed might be unavailable (e.g., behind enemy lines, in toxic areas), or hard to access. Structured deployment might not be possible, and sensors nodes can for instance be deployed by airdrop

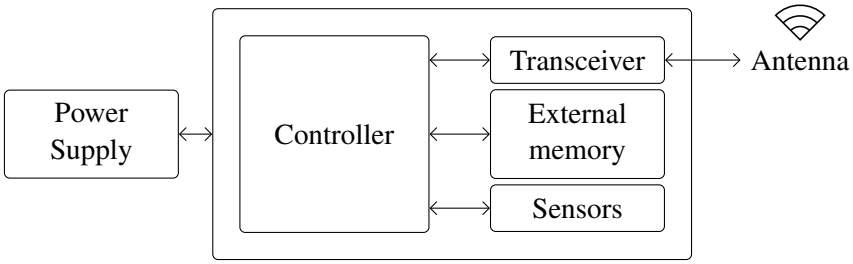


Figure 2.1: "Overview of a sensor node"

from plane or helicopter. In scenarios like this, there is neither any pre-existing infrastructure, nor any structure to nodes placement. A WSN therefore needs to be *self-organized*. Network topology has to be self-established, and be able to adapt to changes in the network. A WSN often contains a base station, also called a root or a sink. The base station might act as a gateway to other networks, or be the destination for the sensor information.

The low price and small size of sensor nodes have advantages, but also drawbacks. Batteries are often the only possible power source for a sensor node, and in many situations recharging or replacement of batteries is impossible. Available power in a battery is dependent on its physical size. The batteries are limited by the size of the sensors nodes, which should often be small. This limits the available power. It is valuable to gather information for as long as possible. The nodes should therefore be energy efficient.

Network lifetime is often regarded one of the most important metrics in evaluating sensor networks. There are many different definitions on lifetime, which are be based on priorities such as: number of nodes that are alive, or sensor coverage [2]. Lifetime improvements can be applied to the operation of single nodes and to the network as a whole. This can for example be done through making nodes sleep when they are not in use, use Media Access Control(MAC) layers that minimizes number of retransmissions, or by using dynamic routing protocols that aims at making the network operate energy efficiently as a whole.

2.2 Sensor nodes

Sensor nodes consist of 4 main parts, a *controller unit*, *sensors*, a *power supply* and a *transceiver* (Fig. 2.1). Each of these components has their own responsibilities.

- *Sensing unit* – Sensors are responsible for gathering information (temperature, pressure, etc.). This information is supplied to the controller.
- *Controller unit* – Controller units processes the data and controls all the components. This is often a microcontroller, but can also be a DSP, ASIC, or FPGA. The controller gathers information from the sensor unit, stores it in the external memory if necessary. Microcontrollers are often limited in processing power. [3]
- *Communication unit* – The communication unit in a WSN is wireless. Several wireless communication standards exists that are suitable for WSNs. WSN communication is usually done by low-cost devices with a low-power usage and a low data rate. IEEE802.15.4[4] is one of these standards.

It specifies a raw data rate of 20-40kb/s in the 868/915MHz bands, and 250kb/s in the 2.4GHz band. Some other bands can also be used. The transmission power in 802.15.4 should conform to local regulations. The common node IRIS states a range of more than 300m outdoors, and more than 40m indoors[5].

- *Power supply unit* – The power supply gives the sensor node power. The power supply can be a main power source, a vehicle with the ability of generating power, or batteries. Batteries are most often the power source used by sensor nodes.
- *External storage* – The external memory can be used to save information that is unable to fit in internal memory. The external memory can also often be kept across reboots and periods of power loss.

A sensor node has, as mentioned, very little resources available due to its small size and the low price. Drawbacks because of this are slow processing capabilities, small memory size, and low amount of available power.

2.3 Base stations

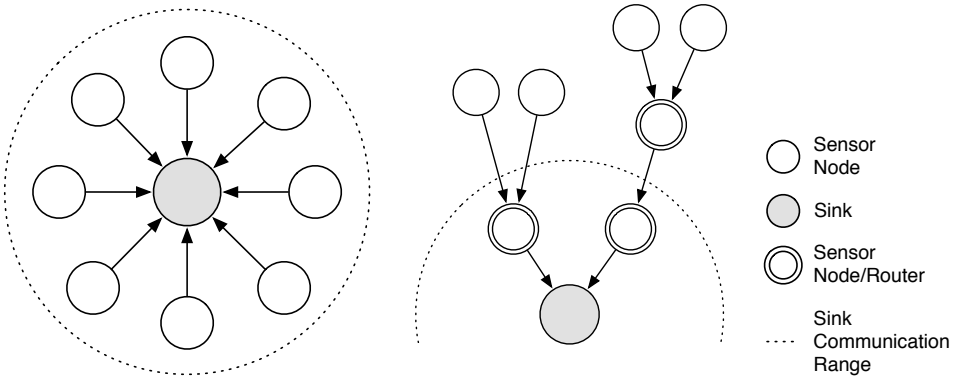
In addition to the sensor nodes, a WSN often contains one or more base stations. Sensor nodes can report the gathered information to the base station, which might process it, store it, and make it accessible to the end user. A base station usually has more resources than a sensor node, making it more similar to a computer. The base station can also be responsible for sending commands to nodes, such as configuration messages or queries. The base station can also be connected to, or act as a gateway, and give the WSN connectivity to other networks (e.g., Internet).

2.4 Data reporting

There are several models that can be used to decide when a sensor node should report the gathered information. Different models are suitable in different applications. Some models are [6]:

- *Time-driven/continuous* – In this model the sensor measures and reports information periodically. This is suitable for situations where periodic measurements are of interest.
- *Event-driven* – Reporting is triggered by certain sensor events in this model. Examples of events are sensor readings that exceed certain thresholds (temperature, pressure), or physical events like a door that is opened or movement in an area. This model fits well in time critical applications as the sensor nodes report events immediately as they happen. Forest fire detection would be an application where event-driven reporting is suitable. A sensor node would react quickly if it observed something that can indicate a fire.
- *Query driven* – Sensor nodes report their readings when they are queried in this model. Queries usually originate from the base station, but can also originate from other nodes. Query driven reporting is suitable when sensor information updates are needed sporadically.

A WSN can use a mix of these different models.



(a) Single-hop (b) Multi-hop

Figure 2.2: Topologies

Information gathered by a sensor node can be communicated to the base station in different ways. The simplest way is for sensor nodes to send the raw-information directly to the base station. There exist techniques that reduce the information transmitted. This can be:

- *Compression* – The sensor nodes can compress the data prior to sending it. Compression is a process that removes redundancy in information, thereby reducing its size without discarding any information. Compression requires some computation, but reduces communication cost, as there is less data to transmit. This increases energy efficiency, as radio transmission often is more power consuming than computation. The effect of the compression is dependent on the redundancy in the data. [6]
- *Cluster aggregation* – Cluster aggregation is a technique where a WSN can be divided into zones called clusters. A sensor node is chosen to be a cluster head in each cluster. Sensor nodes within a cluster report their data to the cluster head. The cluster head can aggregate the information received from the nodes. This can be done by suppressing duplicate data, computing minimum/maximum/average values, percentiles etc. Instead of forwarding the raw data from all nodes, the cluster heads send the aggregated data to the base station. This data size reduction can decrease communication costs [7, 6].

2.5 Topologies and communication

Depending on the positioning and maximum communication distance of the nodes, two different communication scenarios are possible when a network has been deployed. All the nodes can be within direct communication range in a *single-hop* topology (Fig. 2.2a). If nodes are not within direct communication range, intermediate nodes can be used to relay information in a *multi-hop* topology (Fig. 2.2b).

A multi-hop topology is a more complex situation than a single-hop topology. The sensor nodes out of range with the base station have to use intermediate sensor nodes that forward messages to the base station or other nodes. These Intermediate

nodes that forward traffic are called routers. A routing protocol is necessary to make this work. There are several routing protocols for WSNs, with different attributes that are suitable for different applications. Examples of routing protocols for WSNs are RPL, which will be described and used later in this thesis, Anchor Location Service(ALS) and SecRout.

There are 3 types of normal traffic flows in sensor networks. These traffic flows describe how nodes communicate with each other, they are:

- *Multipoint-to-point (MP2P)* – A lot of the traffic in WSNs is MP2P. Many nodes gather information, which is sent to the same destination. This destination is very often the base station.
- *Point-to-multipoint (P2MP)* – Some traffic in WSN can be P2MP. One source is sending information to many destinations. This can be commands, queries, or other messages from the base station to several sensor nodes.
- *Point-to-point (P2P)* – P2P is direct communication from one node to another.

Data collection networks, which will be used in this thesis, mostly consist of MP2P communication flows, where nodes send information to the base station.

Chapter 3

RPL Routing protocol

This chapter introduces and explains the basic working of IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL), a routing protocol created specifical for WSNs. An overview over routing protocols is given in section 3.1. The ROLL working group (WG) is introduced in section 3.2. This WG has created RPL, which is introduced in section 3.3. RPL creates upward and downards routes. The process of obtaining these routes are described in section 3.4 and section 3.5. The trickle timer, which is a central part of reducing energy consumption and network overhead in RPL, is described in section 3.6. Techniques for avoiding and detecting loops in RPL are described in section 3.7. Some repair functionality is also introduced here. Some metrics that can be used with RPL are presented in section 3.8. Finally, the two standardized OFs are described in section 3.9

3.1 Introduction

Routing is necessary for connectivity between devices in a multi-hop network. Routing enables devices to connect to targets that are not accessible locally on any of its interfaces. This is achieved by sending information through other intermediate nodes, called routers. The information, on which routers can be used to reach a given destination, is stored in a routing table. The entries in a routing table can be either for a specific destination, or for groups of destinations divided into subnetworks. These entries can be added to the routing table manually as *static routes*, or dynamically discovered by a *dynamic routing protocol*.

Static routing requires someone to specify all the routes necessary for communication. In order to specify these routes, knowledge of the network topology is needed. Changes in the topology will not be present in the routing tables before someone implements them. Depending on the situation one might need physical access to the devices in need of routing updates. Gaining physical access to devices in a WSN can be difficult, or even impossible to accomplish (e.g., if there are a lot of nodes, or the nodes are physically unavailable).

Dynamic routing protocols automatically discover the network topology, create, and update the nodes routing table according to obtained topology. Networks running routing protocols are self-organizing in the sense that they automatically establish routing information without any manual intervention necessary. The routing protocol keeps running after the network has been

established. It keeps track of changes in the network, and updates the routing information accordingly. In order to keep track of the topology the routing protocol uses control messages that are transmitted on the network. These control messages increase overhead, and in the case of a WSN, increases energy consumption.

Dynamic routing protocols are generally classified into:

- *Interior vs. exterior* – Interior routing protocols are usually working within a network controlled by a single entity, whereas exterior routing protocols work between networks controlled by different entities.
- *Distance vector vs. link-state* – Distance and link vector protocols differ in the way that routing information is stored and exchanged between nodes.
  In a *distance vector* routing protocol a node tells it neighbors about targets it is able to reach and their associated cost. Devices choose the best route by comparing the information obtained from their neighbors. The chosen information is then used to populate the routing table.
  In a *link-state* routing protocol, nodes share information about the targets it is able to reach directly on its interface, with all the nodes in the network. This means that each device has a complete oversight over the entire network. Every link and every node in the network is considered during the calculation of the routes, not only the neighbors.
  A link-state routing protocol has more traffic overhead than a distance vector routing protocol due to the fact that the whole network topology is communicated to all the nodes in the network.

Routing protocols in WSNs and ad-hoc networks are also often classified into proactive and reactive protocols which describe when, and which type of routing information is exchanged

Proactive routing protocols obtain and exchange information about all routes as long as the network is active. Routes to all subnets/nodes are calculated and stored in the routing table. When a node wants to communicate with another node, it already knows how to reach it. Proactive routing protocols work well in situations where large amounts of nodes are communicating. The overhead from discovering and maintaining routes to all the nodes in the network might be unnecessary in networks with sparse traffic.

Reactive protocols, also called *on-demand*, try to obtain information about a route only when it is needed. No other route information is exchanged between nodes. The initiation of communication with a specific node triggers the dynamic routing protocol to discover the needed route.

Reactive routing protocols introduce an initial delay before the packet can be sent. This delay is a result of the time it takes to find a route to the destination. The request might be flooded throughout the network until the destination is found. The path taken by the request message can either be recorded in the request messages, or remembered by the intermediate nodes. When the destination for the route-request receives the request, it sends a reply to the initiator of the route request. The initial delay between a route request and a reply increases with the topological distance between the source and destination. In network situations with much traffic between different nodes, the route discovery process can introduce a lot of overhead, as many request and reply are transmitted for each route that is needed.

It is normal that there is more than one possible route to a given destination. A metric is used by routing protocols to select the preferred route. Examples of

metrics are number of hops, amount of bandwidth, and delay between nodes. The route with the best metric is chosen by the routing protocol.

3.2 ROLL WG and LLNs

"Routing Over Low power and Lossy Network" (ROLL)[8], is a working group (WG) within IETF. ROLL was created Feb. 11, 2011, with the task to create a routing protocol for WSNs for "Low Power and Lossy Networks" (LLNs). LLN is a type of wireless communication technologies with certain characteristics. According to the ROLL charter[8], some of these characteristics are:

- "LLNs operate with a hard, very small bound on state"
- "Optimizes for saving energy in most cases"
- "Typical traffic patterns are not simply unicast flows.""
- "Employed on link-layers with restricted frame-sizes"
- "LLN Routing protocols have to be very careful when trading off efficiency for generality.""

Most networks used in WSN are LLNs. The WSNs described in this thesis use LLNs. The ROLL Working Group (WG) has evaluated already existing routing protocols (e.g., OSPF, IS-IS, AODV, OLSR) for use in LLNs. None of these routing protocols were found suitable for operation in LLNs. ROLL is only focused on IPv6 routing framework in LLNs, and has created "IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL)[9].

3.3 RPL Introduction

IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL) is a distance-vector [10], created by ROLL for use in LLN networks. RPL is a proactive routing protocol, which seeks to minimize routing protocol overhead, at the cost of not always having the latest and most up to date routes. RPL focuses on detecting and creating routes for MP2P and P2MP traffic flows. P2P routes are also supported.

RPL went from being an Internet Draft to a Proposed Standard in March 2012, and is defined in *RFC 6550* [9]. The description of RPL is based on this RFC.

The main routing topology created by RPL is a *destination oriented directed acyclic graph* (DODAG). A graph consists of vertices, which can be connected by edges. A graph like this can be used to describe a network routing topology, where vertices are nodes, and edges are routes. A directed graph means that the routes are unidirectional[1]. The graph being acyclic means that there does not exist any cycles/loops, in the routing between the nodes. The DAGs created in RPL is destination oriented, and all the edges lead to the same destination. In the main DODAG this is the root node.

The DODAG root is the destination in the DODAG. RPL defines two directions of routes, up and down. Routes in the direction toward the DODAG root are *upwards*, while routes going in the direction away from the DODAG root are *downward* routes. The upward routes defines the DODAG. The situation in

[1]Even though a route is unidirectional, it is required that the connection between nodes is bidirectional if they are to be used by RPL

figure 2.2b on page 6 could be a DODAG, with the sink being the DODAG root. Upward and downward routes are detected and calculated using different techniques and messages in RPL. Downward routes are not necessarily created in RPL, this is specified by the *Mode of Operation*(MOP). The DODAG has to be created initially to create downward routes.

It leaves this task to an Objective Function(OF). The OF can use self-chosen metrics in order to achieve its optimization goal. The OF is not specified in RPL, and its behavior might be implementation specific. The OF decides which metric the routes should be optimized for. This makes RPL suitable for WSNs with different requirements. In addition to metrics, RPL can also use constraints (e.g, do not use devices with low remaining energy). The OF is free to also use a combination of several metrics and constraints.

Rank is the result of the OF. It is a number that indicates a nodes distance to root, relative to the rest of the nodes. The root node has the lowest rank in the network. The rank strictly increases as we move down (in the direction from a root towards a node), and strictly decreases as me move up (in the direction from a node towards root). This helps RPL avoid routing loops.

RPL operates through the use of control messages. RPL control messages are ICMPv6 messages, of the type 155. RPL mainly uses 4 types of these messages:

- 0x00 – DODAG Information Solicitation (DIS)
- 0x01 – DODAG Information Object (DIO)
- 0x02 – Destination Advertisement Object (DAO)
- 0x03 – Destination Advertisement Object Acknowledgment (DAO-ACK)

DIS Messages are used to request routing updates, DIO messages are used to establish upward routes, while DAO and DAO-ACK are used to establish downward routes. Secure variants of these messages also exist. The security measures include integrity, replay protection, confidentiality and delay protection.

4 identifiers are used by RPL to identify a routing topology:

- *RPLInstanceID* – Defines which RPLInstance this is. A network can contain multiple RPLInstance. There can only be one OF in a given RPL instance. A node can only belong to one DODAG in a RPL Instance, but can be part of several RPL instances.
- *DODAGID* – There can be one or more DODAGs in an RPL instance, which are identified by the DODAGID. The DODAGID of a DODAG is the IPv6 address of the root node. A node can only be a member of one DODAG at any moment. The scope of the DODAG is a RPLInstanceID.
- *DODAGVersionNumber* – The DODAGVersionNumber identifies a given topology in the RPLInstance defined by RPLInstanceID and DODAGID. The DODAGVersionNumber defines an iteration of a DODAG with a given DODAGID.
- *Rank* – Defines distance from root. The distance is relative to other nodes in the network. The calculation of the rank is done by the OF. It can be a result of link metrics, track topological distance and other properties.

These identifiers above are included in many of the RPL Control Messages. The RPLInstanceID, DODAGID and DODAGVersionNumber must be kept constant as the messages dissipate throughout the network, but the rank field is updated at every node.

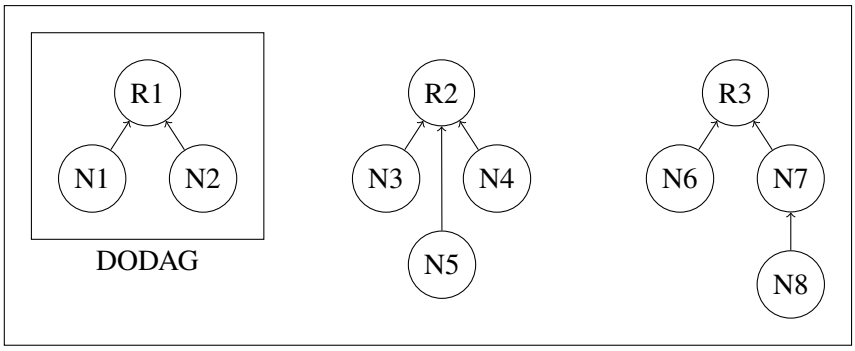


Figure 3.1: DODAGId within an RPLInstance

3 DODAGs within an RPL instance

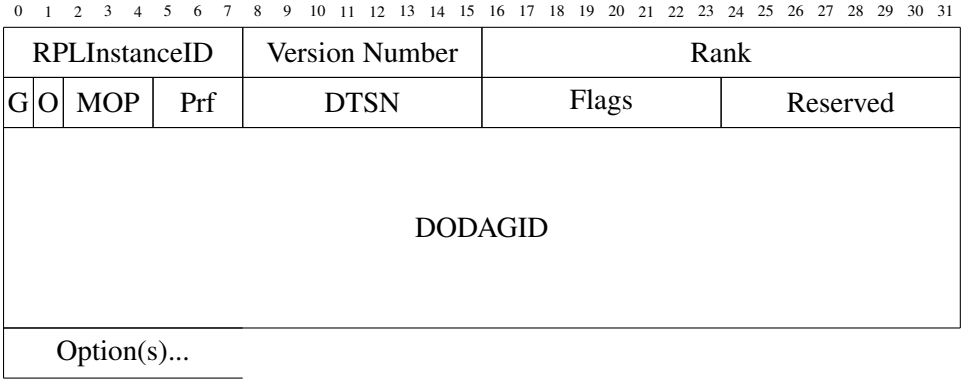


Figure 3.2: DIO Message structure

3.4 Upward routes

Upward routes are used for messages traveling in the up direction, towards the root. In scenarios that needs a MP2P traffic flow, from the nodes to the root, upward routes have everything needed for proper operation. Upward routes are suitable for networks that need MP2P, or P2P connectivity, from nodes to the root.

3.4.1 DIO messages

DIO messages are used to discover and maintain upward routes. The DIO messages are sent periodically. The trickle timer calculates the interval between DIO messages. The DIO messages contain information such as the RPLInstanceID, DODAGID, DODAGVersionNumber and rank. The DIO message format is shown in figure 3.2. All of these options, except for rank, are set by the root node, and should not be changed by any other nodes.

The figure includes some fields that have not yet been described:

- *Grounded* (G): This field indicates whether the DODAG is grounded or floating. If this flag is set, the DODAG is grounded. A grounded DODAG is a DODAG that is able to fulfill an application specific goal. DODAGs that are not grounded are floating.

- *Mode of Operation* (MOP): Indicates the Mode of Operation for the DODAG. This controls the behavior of downward routes. MOP will be described in more detail later.
- *Preference* (Prf): This field defines the preference of the root. This is used to choose between root nodes if more than one exists.
- *Destination Advertisement Trigger Sequence Number* (DTSN): A sequence number used in the procedure of maintaining downward routes.
- *Unused*: The fields *O*, *Flags*, and *Reserved* are currently not in use.

A DIO Message can contain one or more options, the options supported are:

- *0x00 - Pad1*
- *0x01 - PadN*
- *0x02 - DAG Metric Container* - The metric container is used to report metrics for links. The metric can be either aggregate, or discrete link information. This field can be included several times in a DIO message.
- *0x03 - Routing Information* - Contains routing information about reachable networks from the DODAG root. This field can be repeated for multiple destinations.
- *0x04 - DODAG Configuration* - Contains information about different settings used in the RPLInstance. This includes trickle timer configuration values. Variables used for rank computation.
- *0x08 - Prefix Information* Contains information that is used for the IPv6 Neighbor Discovery. This can be used by a node to perform Stateless Address Auto-Configuration (SLAAC) from a prefix advertised by a parent.

3.4.2 Route Discovery

A root initiates the process of discovering upward routes. The root starts sending DIO messages to all its link-local neighbors using multicast. The neighbor nodes receive these messages. If they are running RPL, and the messages are intact upon reception, the DIO messages are processed. Neighbor discovery is accomplished by reception of the DIO messages.

Upward routes are defined by a parent node. A parent is one of the immediate successors of the node on a path towards the root. The preferred parent is used as the next-hop router in the up direction. RPL use three logical sets to calculate the preferred parent. The OF decides which nodes should exist in the three sets.

- *Candidate Neighbor set* – A subset of link-local multicast nodes. The selection of this set depends on the OF and implementation. This is the first selection of nodes that might work as parents.
- *Parent set* – A subset of the candidate neighbor set. Nodes that are in this subset must have a lower rank than the node. All nodes in the parent set must be part of the same DODAG Version.
- *Preferred Parent set* – One node is selected from the parent set, and used as a parent. In the case of multiple parents with identical rank that are equally preferred, there can be more than one node in this set.

A nodes rank is INFINITE_RANK by default. INFINITE_RANK is the maximum rank a node is allowed to have, it indicates that the node does not have any parent.

The reception of DIO messages populate the Candidate Neighbor set. The Candidate Neighbor Set does not necessarily contain all nodes which DIO

messages are received from. The OF uses the Candidate Neighbor set to find parent nodes, which are inserted into the parent set. These parent nodes must have a lower rank than the current rank of the node. Nodes that would cause the parents rank to increase, are not considered in the parent selection process. The node that will give the current node the lowest rank is selected to be the preferred parent. In addition to metric, RPL can also use constraints to restrict certain nodes to be chosen as parents. The constraint can either be applied to link or nodes. The paths containing a constrained link or node will be removed from the candidate neighbor set, and thus never selected to be a parent. Examples of constraints can be to not use nodes with low remaining power as routers, or to limit the maximum amount of hops. Battery operated nodes will not be chosen as parents. Messages will not be routed through any battery-operated nodes.

When a node has selected a preferred parent and calculated its own rank, it starts multicasting DIO messages to its link-local neighbors. As stated, the fields in these DIO messages must identical with the received fields from parent nodes, with the exception of the rank field, which must be updated at every node. New nodes now receive DIO messages and the process of selecting a parent node repeats in all of them. Eventually, upward routes will have been established throughout the network.

A node can use a DODAG Information Solicitation (DIS) to request DIO messages from other nodes. The DIS messages can specify criteria that should be fulfilled for a node to reply with a DIO.

3.5 Downward routes

Downward routes are used for messages in the direction away from the root. The downward routes are necessary in applications using P2MP, or P2P traffic flows where messages are destined to other nodes than root. RPL specifies 4 different Mode of Operations (MOP) that decide how downward routes are handled in the network. The root decides the MOP. There can only be one MOP in a RPL instance[2]. If a node doesn't meet the requirement for acting as a router with the active MOP, it must join the network only as a leaf node. A leaf node uses RPL to calculate parent routes, but cannot take part in routing and act as a parent for any other nodes. RPL specifies 4 different MOPs:

- 0 – No downward routes maintained by RPL
- 1 – Non-Storing Mode of Operation
- 2 – Storing Mode with no multicast support
- 3 – Storing Mode with multicast support.

Downward routes are established and maintained using Destination Advertisement Object (DAO) messages. This has to be done after the DODAG have been created. A node is responsible for transmitting DAO messages for all targets it is associated with. Targets are nodes/networks reachable through that node. DAO messages behave differently depending on which MOP is active. The main difference between DAO in the non-storing and storing MOP, is the location of routing information. DAO messages flow in upward in the network, the opposite direc-

[2]The RPL RFC opens for the possibility of mixing MOPs in an RPL instance in future extensions[9].

tion of DIO messages. The DAO messages contain information about targets on the node, and how they can be reached (transit information). This information is included in the DAO as a target option, possibly followed by a transit information option. The DAO messages might be sent upon reception of other DAO messages, changes in a neighbor set, or the expiration of related prefix lifetime. If the received DAO message is not "new", the node is not required to transmit a DAO. The definition of a new DAO differs in the different MOPs. The DAO messages also contain a Path Sequence counter to indicate freshness. If a DAO with new/different information is to be sent, the node must update its `DAOSequence`.

Point-to-point communication is accomplished by sending the message upward to a router with the necessary routing information, this router then sends the message downwards to the destination node.

3.5.1 Modes of Operation

No Downward routes maintained by RPL (0): In this case, no downward routes exist in the network at all. Nodes should not transmit any DAO messages, and are free to ignore received DAO messages.

Non-Storing Mode of Operation(1): In non-storing MOP, downward routing information resides only in the root node. DAO messages are sent directly to the route as unicast messages. The root keeps track of the DAO messages it has received, and calculates the routes provided by the DAO. If the target address is derived from a prefix owned and advertised by a parent, this can be used as the transit information sent in the DAO. If this is not the case, the transit information is the parent address of a reachable target. All DAO messages in non-storing mode are considered "new".

The routing information in the root consists of nodes and their parent's address, or the prefix if the target address is derived from a parent prefix. The root recursively uses this information to find the route needed to reach the node. Routing operations in intermediate nodes are performed using source routing. The message header contains the route for the messages destination. Each router checks the header, and sends the message to the node appearing after itself in the message.

For two nodes to communicate together in this mode, the transmitting node has to send the message to the root, which uses source routing to send the message on to the receiving node. Source routing is a technique where the complete path from a source to a destination is stored in the message header. Nodes forward messages using this information rather than a routing table in memory. The next-hop address is found by looking up the nodes id in the path, and then read the next address in the list.

Non-storing MOP is suitable in applications where the nodes have limited memory. The nodes do not have to store any routing information. The source routing information in the header might lead to more overhead than in a storing MOP. Because of the maximum header message size, there is a maximum amount of hops the message can traverse. If two nodes are topologically close, messages between these might travel unnecessarily long distances since it has to go via a root.

Storing Mode of Operation, multicast and no multicast(2,3): In storing mode, each router node stores downward routing information for nodes that are beneath it (down direction). DAO messages are either sent to parent nodes as unicast (no multicast), to all link-local RPL neighbor as unicast (multicast). The information in a DAO message contains all targets reachable downwards through this router, including targets more than one hop away.

When sending a message in this MOP, the routing table is first checked for a match against the destination node. If no match is found, the message is forwarded to the parent node. This continues until a match is found, or the message has reached the root node. If an intermediate node finds a match in the routing table, the message is forwarded to this next hop. This way messages first travel to the closest common parent along the line up toward the root. This parent should have a routing entry for the destination, and forward the message in the down direction towards the destination away from root.

Storing MOPs require that nodes have sufficient memory to store routing information. If this is not the case, routing information may be lost. This can lead to dropped messages.

Storing mode requires more memory than non-storing mode. Nodes in WSN often have very limited memory, which might make storing mode hard to accomplish. At the cost of memory, storing mode can reduce energy consumption compared to non-storing mode. Messages only have to find the common parent before the message can be redirected down. This decreases the number of retransmissions needed by at least the amount of hops between the common node and root. The reduction in transmissions also decreases end-to-end delay.

A DAO is considered "new" if:

1. it has a newer Path Sequence number
2. it has additional Path Control bits
3. it is a No-Path DAO message that removes the last Downward route to a prefix. A no-path DAO message is a message clearing the downward routing information created from DAO messages.

3.6 Trickle Timer

The trickle timer is designed to reduce network overhead caused by RPL control traffic. If routing information between nodes differs, the network is in an inconsistent state. This can be fixed by transmitting DIO messages, which updates other nodes routing information. Rapid transmission of DIO messages makes this process faster. If the network is consistent, that is, the routing information between nodes do not differ, rapid transmission of DIO messages creates unnecessary control traffic.

The trickle timer dynamically adjusts the interval between DIO messages. If the network is in an inconsistent state, the interval between DIO messages is small. As long as no inconsistencies are discovered, the trickle timer increases the interval between DIO messages exponentially. The trickle timer is specified in RFC6206[11].

The trickle timer is configured with 3 parameters. A *minimum* and *maximum* interval size, and a *redundancy* constant. The redundancy constant controls how

many DIO messages to send, depending on received consistent DIO messages. The higher the value, the less DIO messages are sent. The minimum and maximum interval size controls the smallest and largest interval allowed to use between DIO messages.

The RPL RFC provides some default values for the trickle timer. `DEFAULT_DIO_INTERVAL_MIN=3` is used to calculate the minimum interval. The default value of 2 specifies the interval to be 8ms($2^{DIOIntervalMin}$). The default maximum interval is calculated from `DEFAULT_DIO_INTERVAL_DOUBLINGS=20`, this makes the maximum interval 2.3 hours ($2^{DIOIntervalMin} * 2^{intervaldoubling}$ms).

DIO messages that do not cause any change to the parent set, preferred parent set, or rank, are considered consistent. Events mentioned in the RPL rfc[9] causing the trickle timer to reset are:

- When a node detects an inconsistency when forwarding a packet.
- When a node joins a new DODAG Version (e.g., by updating its DODAGVersionNymber, joining a new RPL Instance)
- When a node receives certain types of DIS multicast messages (specified in the rfc).

3.7 Loop avoidance and detection

In a LLN, connectivity often undergoes changes. Keeping the routing information updated at all times can require a lot of energy. Traffic is often infrequent, and updated routing information is not always necessary. For these reasons RPL does not keep the routing information updated at all times. Changes in the network that are not updated in the routing tables might therefore lead to loops. RPL does not guarantee that routing loops never occur, but tries to avoid them. To avoid routing loops, RPL expects an external mechanism to include some RPL control information that enables it to detect loops.

3.7.1 Rank

The concept of rank is used to avoid routing loops during the creation of the DODAG. The rank strictly decreases in the up direction. There are several rules that the rank must follow, some of the main ones are:

1. A node must not advertise a rank higher or equal to any of the nodes in its parent set
2. A node can advertise a lower rank than it has previously advertised.
3. A node is not allowed to move down and advertise a rank higher than a given amount compared to what it has already advertised in the same DODAG version. The highest allowed rank is given by $lowest\_advertised + DagMaxRankIncrease$. An exception to this is the INFINITE_RANK.

The first rule ensures that the parents of a node always are closer to root. The second rule makes it possible for a node to improve its place in the DODAG. The third rule prevents greediness. Nodes are greedy by increasing their rank in order to get a bigger parent set. An instability might occur if nodes are allowed to be too greedy. Because of this nodes are only allowed to increase their rank a certain amount. A route can poison its neighbor by advertising INFINITE_RANK. It is

then considered to have left the DODAG, and is removed from the parent of the nodes in its sub-DODAG. A node is allowed to move down, thus decreasing its own rank. There is no danger for routing loops in this scenario.

3.7.2 Loop detection

The rank is also used to detect loops during normal operation. This is done using the RPL Packet Information mechanism. The RPL Packet information contains some information that is to be included with normal data-messages forwarded by routers in the RPL network. The RPL packet information contains:

- Down'O' 1-bit: This flag indicates the expected direction of the packet, either up or down. It is set when the packet is expected to travel in the down direction.
- Rank-Error 'R' 1-bit: This bit is set if a node detects an inconsistency between the expected direction indicated in the packet and the relationship between sender and receiver Rank.
- Forwarding-Error 1-bit: If set, indicates that the router is unable to forward the packet further towards the destination.
- RPLInstanceID 8-bit: The RPLInstanceID that the sending node is a part of.
- SenderRank: This contains the sender's rank. It is updated at every router it traverses.

RPL Packet Information is to be included with all data messages, and contains control information. The control information includes the sender's rank and a flag (called down), which indicates the direction the packet is expected to be routed. The direction indicated by the relationship between sender rank and current rank, is compared with the expected direction indicated in the down flag. A mismatch occurs if the down bit indicates one direction, and the relationship between the sender's rank and current node's rank indicates the opposite direction. If a mismatch occurs and the Rank-Error field is clear, the router sets the rank-error field and forwards the packet. If a mismatch occurs and the Rank-Error field already is set, the router drops the packet. The node should also start a local repair operation (described in section 3.7.5).

3.7.3 DAO loops

DAO loops may occur if DAO messages are lost. Due to the likelihood of this happening, DAO acknowledgment (DAO-ACK) messages exist to make sure DAO messages reach their destination.

3.7.4 Global repair

Global repair is a mechanism that can be used to refresh the DODAG. It lets all nodes select new parents and advise a new rank, independently from what have been before. Only the root node can initiate a global repair. It does this by incrementing the version number. As nodes receive DIO messages with the new version number, they are free to join the new version when they want. However, all their parent nodes must also be part of the new version. When a node has joined the new DODAG version, it sets the new version number in its DIO messages, and

the DODAG is updated node by node. A node that joins a new version has no restrictions to its rank or its parent set, except that all nodes in the parent set must be part of the same version.

A global repair can be triggered by the root periodically, on detected events, or manually.

3.7.5 Local repair

RPL allows local repair mechanisms. One of the suggested methods allows a node to poison its sub-DODAG by advertising `INFINITE_RANK`. Nodes that advertise `INFINITE_RANK` cannot be used as parents. The nodes in the sub-DODAG therefore have to remove the node from their parent sets. The node is now detached from the DODAG, and can choose parents that previously were higher ranked.

3.8 Metrics

Many different metrics can be used for path calculation with RPL. These different metrics are specified in RFC6551[12]. Metric types in RPL can be used as either constraints or metrics, and many of them can be used as both. A constraint limits the selection of parents (e.g., don't choose a parent which routes involves a parent with low remaining power), while a metric is used to optimize a selection.

The metrics included in DIO messages can be *aggregated* or *recorded*. An aggregated metric is defined by one value, which is adjusted as the DIO messages travel through the network (e.g., hop count, which increases by 1 for every hop). Recorded metrics are specified by several distinct values, which are recorded as DIOs travel through the network. The possibility for several metric fields in DIO messages allows the OF to use several metrics. In this scenario a precedence field is given with the metrics, this field decides which metric should have precedence over the others.

RFC6551 includes some often-used metric that can be used:

- Hop Count - Reports the number of traversed nodes along the path.
- Link throughput
- Latency
- Link reliability
- Link Quality Level (LQL)
- Estimated Transmission Count(ETX)

3.8.1 Hop count

Hop count is a very common metric that is widely used in computer networks. The hop count measures the amount of nodes a message has to traverse in order to get to the destination. When used in RPL, hop count can be aggregated or constrained. Hop count can be used either as a metric, or a constraint.

3.8.2 Link Reliability

Two metrics that indicate Link Reliability is specified for use with RPL. These are *Link Quality Level(LQL)* and *Estimated Transmission Count (ETX)*.

LQL is a number ranging from 0 to 7 describing link reliability. A value of 0 indicates unknown link quality, a value of 1 indicates the highest link quality, and a value of 7 indicates the least link quality. This path metric can be used as either a constrain or a metric. The conversion to the LQL is not specified, and is implementation specific.

ETX is the other link reliability metric that can be used with RPL, it can be a path metric or a constraint. ETX[13] was designed to overcome some of the problems that occurred with hop count in lossy networks. The ETX metric indicates the total amount of expected transmissions needed for a packet to be successfully received and acknowledged. It therefore also accounts for link asymmetry as both directions are considered. ETX can be calculated using forward and reverse delivery. There is no requirement for this formula, but the following formula is often used:

$$ETX = \frac{1}{d_f * d_r}$$

where d_f is the forward delivery ratio, and d_r is the reverse delivery ratio. The first ETX proposal[13] used network probes to gather this information. These probes are not very suitable for WSN networks, as they introduce unnecessary network overhead. Another way often used to measure the ETX is to give all nodes an initial ETX, which is often a bit high. A random node is initially chosen as the preferred parent. As this route is used, the ETX is updated. Two RPL implementations, TinyRPL and ContikiRPL, update the ETX with a exponentially weighted moving average (EWMA). This technique finds errors and corrects for them. A weakness using this method is that nodes that can provide a better route is not necessarily tested. A similar method called passive probing [14] has been proposed to improve this. Passive probing technique uses a low initial ETX value, and switches the preferred parent often. The fast switching would allow many preferred neighbors to be tested, increasing the chance of finding the best parent.

The ETX metric has some important characteristics. Throughput is positively affected by the fact that ETX is based on delivery ratios. ETX avoids routes with many hops, as each hop increases the ETX with at least 1. This might have the effect of reducing energy consumption, as each packet transmitted uses power both for the receiver and transmitter.

ETX offers improvements over hop count in several areas. It minimizes the amount of hops, while also accounting for loss ratios and asymmetric links. This can also reduce energy consumption pr packet. Link asymmetry is also accounted for as delivery ratios in both directions are considered.

3.9 Objective Functions

The Objective Function (OF) is responsible for selection and optimization of routes. The OF is an interchangeable part of RPL, and RPL does not specify an OF. Users can create their own OF if they want to. This separation makes it easy to tailor RPL for different use cases, as different OFs can have different optimization goals. An OF can specify the metrics and constraint it uses. The OF is not an algorithm, but an abstraction as to what it should do. One OF can optimize routes based on latency, while another OF can optimize routes based on

energy consumption, and constrain the routes to use nodes that are not low on power.

The OF uses information collected from the DIO Dag Metric container option, as well as the rank.

Objective Function 0 (OF0) [15] is the default OF in RPL. It is assumed that OF0 is always implemented. OF0 is a standard specified in RFC6552.

3.9.1 Objective Function 0

Objective Function 0 (OF0) is the default OF in RPL. Its goal is to provide good enough connectivity to a grounded DODAG, but does not guarantee optimization against a specific metric. OF0 uses the rank information from DIOs, but not the metric container. OF0 selects a preferred parent, and a feasible backup successor if possible. The feasible backup successor is only used if connection to the preferred parent is lost.

OF0 uses `rank_increase` to calculate rank. `rank_increase` is a normalized value. `step_of_rank` is a property associated with a link to a node. `step_of_rank` is a number between 1 and 9. It is recommended that `step_of_rank` is based on link-quality. If `step_of_rank` is constant, the rank will be similar to hop count. This will result in longer hops, with added risk that the link is bad.

Although not recommended, OF0 allows `step_of_rank` to be stretched by `stretch_of_rank`. This is done using a `rank_factor`, to which `step_of_rank` is multiplied. `rank_factor` can be used to give certain link types priorities over other. The resulting rank_increase for a node is given in the equations:

$$\begin{aligned} rank\_increase &= (rank\_factor * step\_of\_rank + stretch) \\ &\quad * MinHopRankIncrease \\ resulting\_rank &= rank\_parent\_node + rank\_increase \end{aligned}$$

`MinHopRankIncrease` is a number defined in RPL. It decides the minimum rank that must be added to a rank, and also the radix point in the rank value. [15]

Parent selection

OF0 compares nodes based on the following criteria:

1. The reachability of a node mus be validated before it is selected as a preferred parent.
2. Consider policies for multiple interfaces and administrative preferences.
3. Connectivity to a grounded root should be preferred.
4. Connectivity to a more preferable root should be preferred.
5. For nodes in the same version, the most recent version should be preferred.
6. The node that causes the lower rank should be preferred.
7. (Optional) A node where a feasible successor exists should be preferred.
8. The previously selected parent should be preferred.

9. The node that has announced a DIO message more recently should be preferred.

3.9.2 Minimum Rank with Hysteresis Objective Function

The Minimum Rank with Hysteresis Objective Function (MRHOF)[16] is the second current standardized OF for RPL. It is defined in RFC6719. MRHOF chooses parent nodes trying to minimize a given metric. The metrics to be used with MRHOF must be additive (e.g., hop-count, ETX). MRHOF also uses hysteresis to minimize changes due to small metric changes. MRHOF can work with several different metrics, and uses the metric received in the DIO Metric Container. ETX is used as a default if no DIO Metric Container is present. ETX is then sent in the rank field of DIO messages.

In the parent selection process MRHOF searches for the parent that provides the lowest end-to-end cost, if this new parent offers a route that is better than the currently selected route by a given threshold, it will be selected. If this threshold is not exceeded, MRHOF stays with the old parent. The threshold is the hysteresis part of MRHOF.

MRHOF also specifies certain rules for calculating rank when the parent set size is larger than one.

Parent selection

Path cost is calculated for each candidate neighbor reachable on an interface. A path cost is received from each neighbor. This is the path cost from that node, to the root node. The calculation of a path cost to a neighbor is a sum of the path cost advertised by that neighbor, and the cost from the calculating node to the neighbor node as shown:

$$path\ cost = advertised\ neighbor\ path\ cost + cost\ to\ neighbor$$

The neighbors' path cost should be updated when:

1. The selected metric of the link to the candidate neighbor is updated.
2. The selected metric is a node metric and the node is updated.
3. A node receives a new metric advertisement from the candidate neighbor.

When all the path costs have been calculated, MRHOF chooses the preferred parent based on the path cost. Nodes whose path cost is larger than `MAX_PATH_COST` should not be . The variable cur_min_path_cost, is the path cost of the currently preferred neighbor. It is used with `PARENT_SWITCH_THRESHOLD` for to allow hysteresis. MRHOF allows a node to select `PARENT_SET_SIZE -1` candidate neighbor in its parent set. `MAX_LINK_METRIC` is used to filter out bad links. If the path cost to a neighbor node > `MAX_LINK_METRIC`, this node should not be considered.

The advertised rank when using MRHOF depends on the metric used. If hop-count or ETX is used, the rank is equal to the cost. If latency is used, the rank is $latency/cost$. If `PARENT_SET_SIZE`>1, the maximum rank calculated from the 3 following rules is used

1. The rank calculated for the route through the preferred parent.

2. The rank of the member of the parent set with the highest advertised rank, rounded to the next higher integral rank.
3. The largest calculated rank amongst paths through the parent set, minus `MaxRankIncrease`.

MRHOF also proposes some recommended values based on experience from real life deplyments[16, 17].

Chapter 4

The IRIS node

This chapter introduces the IRIS sensor node, and some accessories that, be used for the work of this thesis. An overview of the node is given in this chapter. The board for connecting the nodes to a computer is thereby given, and the sensor board that will be used later. In the last section the radio and its link quality indicators are described.

4.1 IRIS

The sensor node that is used in this thesis is the Memsic IRIS node[5]. This is a node that is supported by the standard TinyOS version. A picture of the IRIS node with a MTS420 sensor board attached is shown in figure 4.1. The IRIS has 3 user programmable LEDs, and a 51 pin expansion connector. This connector can be used for analog inputs, digital i/o, I2C, SPI and UART interfaces. The node has a size of 58x32x7mm excluding the batteries, and weighs 18g, and is powered by two AA batteries at 1.5V each.

The processor on the IRIS node is XM2110CB, which is based on the ATMega1281. The node is equipped with 8KB of RAM, 128KB of programmable flash memory, and 512KB of measurement flash. The processor uses 8mA in normal operation, and 8µA in full sleep.

Figure 4.1: The IRIS node, equipped with a MTS420 sensor board

4.2 UART board

The MIB520[18] enables all IRIS nodes to be used as gateways. The MIB520 connects to the expansion connector of the IRIS. Attached to the MIB520 board are LEDs connected showing the same as the LEDs on the node. The connector also provides the node with power, and enables programming and UART communication with the node through a USB port, with a baud rate of 57.6K. The MIB520 will be used for the base station, and also to program all nodes.

4.3 Sensor board

The sensor board MTS420[19] is a sensor board designed for the IRIS, and other similar nodes. The sensor board connects directly to the expansion connector on the IRIS. It is a collection of multiple sensors, which amongst others can measure temperature, humidity, dual-axis acceleration, barometric pressure, and ambient light. It is also possible to equip this sensorboard with a gps module.

The Sensirion Sht11[20] is the sensor used to measure temperature and humidity on the MTS420. This sensor offers a minimum resolution of 0.4%RH and 0.04°C. The sensor operates with voltages between 2.4 and 5.5V. The SHT11 consumes 0.3μA when sleeping, and 0.55mA when measuring.

The operating range of the sensor is -40–123.8°C, and 0–100%RH, with a typical humidity accuracy of ±2.0(%RH), and a typical temperature accuracy of ±0.4°C. The accuracy of the sensors can be affected when operating under extreme conditions, e.g., larger than 80%RH.

4.4 Radio

The IRIS node is equipped with an IEEE802.15.4[4] compliant radio, Atmel RF230[1]. RF230 operates between 2.4 and 2.48GHz with a data transmission rate of 250kbps. Maximum transmission power is 3dBm, with a stated transmission range of more than 50m indoor, and 300m outdoor. Transmission at this power output consumes 17mA. Power consumption can be reduced by reducing power output, this results in a power consumption of 13mA@-3dBm, and 10mA@-17dBm. The radio has a receiver sensitivity of -101dBm. The minimum received power indication is however only -91dBm.

4.4.1 Link quality

The RF230 radio offers access to 3 properties that can be used to indicate the quality of a received message. These 3 properties are *Receiver Energy Detection* (ED), *Received Signal Strength Indicator* (RSSI) and *Link Quality Indication* (LQI), and are all specified IEEE802.15.4[4]. These properties can be read when receiving packets.

RSSI and ED

RSSI and ED are both used to measure channel energy for received packets. RSSI and ED cannot be used simultaneously. RSSI indicates the channel energy at a

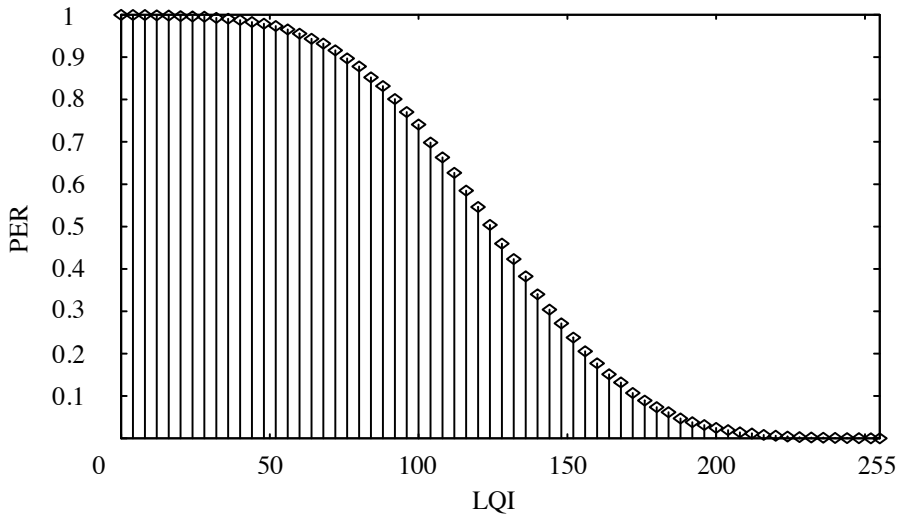


Figure 4.2: Conditional Packet Error Rate versus LQI. (From [1])

This figure shows the association between LQI and PER for the RF230 radio. These are statistical values based on a large number of receptions for frames with a length of 20 octets.

given instant in time. RSSI has a minimum sensitivity of -91dBm, and a dynamic range of 81dB with a resolution of 3dB. ED is an average of RSSI over 8 symbols (128µs. This gives it a higher resolution than RSSI. ED thus has a resolution of 1dB, and a minimum sensitivity of -91dBm.

LQI

LQI is a value ranging from 0 to 255, and is given as a combination of signal strength and/or qualify of a received packet. IEEE802.15.4 leaves the calculation of LQI to the implementation, given the conditions that 0 indicates lowest link quality, and 255 indicate the best link quality. The values should be distributed evenly between them. RF230 uses correlation results of multiple symbols within a frame to determine the LQI. The LQI calculated from RF230 can be associated with Packet Error Rate (PER). This association can be seen in figure 4.2. Low values are associated with low signal strength and/or signal distortions. These distortions can either be caused by interference and multipath propagation. The LQI grows independent of RSSI/ED the larger the RSSI/ED is, since this does not improve Packet Reception Rate (PRR).

Chapter 5

Choosing metric and OF

A big part of a RPL implementation is the Objective Function. RPL[9] specifies that Objective Function 0 (OF0) must be implemented. OF0 is to be used as the least common denominator for RPL implementations. The RPL implementation used in this thesis does not need to be interoperable with any other implementations. The implementation of OF0 was not deemed necessary. In addition to choosing an OF, metrics to be used also have to be chosen

This chapter starts by explaining the needs for the application. Thereafter some experiences about metrics from test deployments and experiments are discussed in section 5.1. This leads to an optimization goal and use of metric that is described in section 5.2. Finally, an OF is chosen in section 5.3

5.1 Experiences with metrics

Different use cases have different needs for the metric that should be used in the routing protocol. Since RPL is not metric specific, one can choose between many different metrics. RFC6551[12] specifies metrics that can be used with RPL.

The application that this metric was provided to support was a data-collection tree, where all nodes gather information, which is transmitted to the root node. A network with a long lifetime and stable communication was wanted. Routing optimization based on nodes remaining energy was out of the scope for this implementation, but the routing protocol should provide reasonable energy efficiency pr packet. Some wanted characteristics were created for the metric:

- The selected routes should be energy efficient in that they do not have an unnecessary amount of retransmissions. Routes with an unnecessary high amount of hops should also not be selected. Routing protocol overhead should also be minimized in order to save energy.
- Combined with the previous point, the chosen routes should be reliable to a given point. The amount of maximum retransmissions should thus be kept within a certain amount.
- End-to-end latency was not very critical for this network. The network would gather information with intervals larger than 5 minutes. These numbers are much larger than the end-to-end time in most networks.

Some of the metrics introduced in section 3.8 were considered for accomplishing these tasks.

5.1.1 Hop count

Hop Count is a metric that is very much used in computer networks, where it works well. The amount of hop is minimized, without taking regard to bandwidth or loss. The difference between computer networks and LLN networks is however big. LLN networks often vary much, and can be very lossy. Hop count does not work too well in LLNs, as link quality is accounted for. Hop count therefore often chooses a preferred parent that is far away, which increases the chance that a link is lossy[15, 13].

Testing hop count

A test was done using a basic hop count implementation in Bergen University College. This implementation minimized hop count, and used node id as a tiebreaker. Given nodes with equal hop count, the node with the lowest node id was preferred. The results from this test showed poor results for the hop count metric.

A network of about 10 nodes with the hop count optimization was deployed. These all sent information to the root with 1-minute intervals. Most of these nodes were located above lowered ceilings, with groups of nodes being located in different rooms. . Either plaster or concrete walls separated the rooms. Not given much consideration, nodes were placed with increasing id numbers further and further away from the base station. The node id thus increased as distance from root increased. The nodes closest to the root node showed a bit higher packet loss than expected in this experiment. This is most probably due to much WiFi traffic, which is around the same frequency area as the IRIS node. The locations of the nodes might have also affected this. Most of the nodes chose the base station as a preferred parent. This is as expected since it offers the lowest hop count, and also has the lowest node id. This worked well for nodes that were close to the base station. High packet loss did occur for nodes that located further away. It was also observed that almost all nodes selected the root node as a preferred parent, even if packet loss was very high. This resulted in a couple of received packets from some nodes.

The experiences from this test led to the following conclusions:

- Since hop count minimizes the number of hops, it often chooses parents that are far away compared to the nodes within communication distance. As link strength, and also quality, is largely a product of distance, this increases the chance of selecting a parent with one of the worst routes..
- The previous point was made even worse by the fact that the node id increased with distance, and the implementation chose the parent with the lowest node id.
- Even if a link is very lossy, some routing messages might make it through from time to time. The routing protocol does not know that these links might be very lossy, and might see a good candidate.

These results fit well with results from other literature [13].

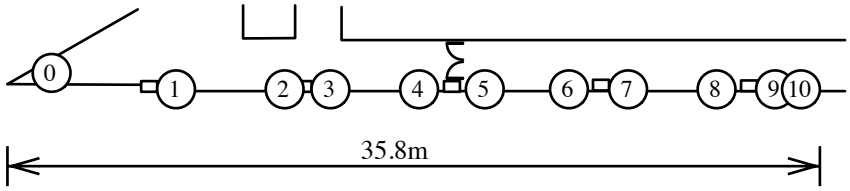


Figure 5.1: Test node locations

5.1.2 RF230 Link Quality

An experiment was done in order to get some experience with the link quality information provided by the RF230,packet loss, transmission distance and link asymmetry. This information would prove valuable information in selecting a metric, and when deploying a network. Some of these properties are location dependent, and will therefore not be the same in different locations, they do however give a certain indication to the relationships. Examples of these properties can be noise, which can greatly affect link quality and asymmetry.

The test consisted of 10 nodes in addition to a base station. The nodes were placed throughout a hallway, following a straight line. An overview of this can be seen in figure 5.1. The test was done in a weekend, the environment was therefore mostly static as there was not much human activity in the hallway.

The rest of the 10 nodes were spread evenly throughout the hallway, with an increasing distance from the base station. Node 10 was furthest away from the base station, and was placed where packet reception seemed to stop. The hallway was about 2m wide. In order to keep the nodes out of harms way, they were placed on lockers by the wall. Some structural hindrances were located between the nodes and the base station, and some nodes were located in spots where radio shadows where likely to occur.

During the test period, the base station would in turn send a ping message to each node containing a sequence number. A reply would be sent to the base station if a node received a ping message like this. The reply contained the sequence number, and also ED and LQI from the ping message from the base station. Upon receiving this reply the base station would log the nodeid, sequence number, and ED and LQI in both directions. Table 5.1 shows an overview of the results from this experiment.

Transmission distance and PRR

From the results in table 5.1 we can see that PRR somehow decreases as distance increases. Node 7, 8, and 9 shows that these factors are not necessarily directly connected. Node 9 has a higher PRR than both 7 and 8, even though it is further away. We can also see that node 8 has a very low PRR. This is most likely due to multipath fading, or other external factors. The nodes seems to be divided in 3 different regions: a *connected*, a *transitional*, and a *disconnected* [21] region. The connected region consists of links that are stable and of good quality. Link quality in the transitional region is often varying, and is not a product of distance. There is also often high asymmetry in the transitional region. The disconnected region consists of links that have very poor quality, and is mostly unusable. In the

node id	distance(m)	PRR(%)	avg. ED (σ)	avg.LQI (σ)
1	5.5	99.87	-72.0 (0.50)	255 (0.94)
2	10	86.18	-74.4 (0.29)	255 (4.54)
3	12	87.82	-82.0 (0.14)	255 (7.55)
4	16	83.25	-82.5 (1.30)	255 (2.13)
5	19	83.68	-90.0 (0.25)	252 (14.0)
6	23	87.94	-76.0 (0.12)	255 (3.5)
7	25	78.49	-90.6 (0.48)	236 (23.0)
8	29	29.41	-91.0 (0.06)	142 (26.4)
9	32	84.06	-90.5 (0.5)	237 (27.0)
10	32.2	$6.9 * 10^{-3}$	x	x

Table 5.1: Results from the hallway test

This table shows an overview over the different nodes and results from the test in the hallway. It includes the distance from root, PRR, average RSSI with standard deviation, and also average LQI with standard deviation. Some data is excluded from node 10 as there is not enough to make any statistics (A total of 4 messages were replied).

connected region the communication is stable, node 1-6 in our test would be in the stable region. The transitional region contains node where communication can differ, studying the packet reception to node 8 and 9 shows this. Node 8 has very poor connectivity, but node 9 has good connectivity. Barely any communication has been accomplished with node 10, this node is in the disconnected region.

LQI/ED and PRR

LQI and ED/RSSI are the two link quality properties accessible in the RF230 radio. Unfortunately it was not easy to get the value of the noise floor in TinyOS, and it is therefore not studied. It is of great interest to figure out how LQI and ED/RSSI is related to PRR. According to the RF230 data sheet [1] there is a large correlation between LQI and PRR, but not so much for ED/RSSI.

Figure 5.2 shows PRR vs average LQI and PRR vs average ED from the experiment. T. This graph does not show much correlation between ED and PRR. Low PRRs would be expected close to the radio minimum sensitivity [22]. I would however think that this would be highly affected by the signal to noise ratio. The minimum ED detection sensitivity is however 10dB over the minimum receiver sensitivity of the radio, a measurement of -91dBm can be anywhere between -101dBm and -91dBm. Packet loss close to this region can therefore not be seen.

Figure 5.2 does show a large correlation between LQI and PRR. This is similar to the information provided by the RF230 data sheet [1], but the results shows a higher needed LQI for a given PRR. This is expected as the PRR in this test is a product of both directions, the frame size is also larger in this experiment than the data sheet. The graph is kinda inconclusive, as there is not much data for LQI between 170 and 220. Note that LQI is fluctuating fast, and therefore needs to be averaged [22], this can be seen in figure 5.3 where differences in LQI of 100 is experienced during a period of 30 seconds.

Asymmetry in radio links for LQI and ED can be seen in figure 5.4. There seems to be a very strong symmetric relationship between sent and received ED. The LQI shows more signs of asymmetry. Note that the color intensity of this graph is logarithmic, and areas with a low amount of observations might seem like has a larger amount of observations.

The standard deviation shows that there are low variations in ED for the nodes, but a bit higher variations for LQI.

5.2 Choosing optimization goal and metric

The discussed metrics, and the results from the test and experiment were taken into account when selecting a metric to use. The hop count test showed poor results, according to the expectations. Using hop count as a metric was therefore not wanted. The ETX metric met many of the wanted characteristics in this implementation. The described ways of finding the ETX was however not very appealing. Probes increase network overhead, a high initial ETX has the possibility of not finding good parents. The low initial ETX value could work, but the implementation could have long intervals without traffic. This would mean that the ETX information could be old by the time all parents were tested.

The experiment done in the hallway showed a large correlation between LQI and PRR, which was also confirmed by the radios documentation. Since ETX is a product of PRR, the LQI could be used to make an estimated LQI. The estimated ETX would not be able to measure the asymmetrical properties of the link, but could provide a good initial value. This implementation used only the LQI for ETX calculation. In order to get real results, the real PRR of the active links could be used, providing a correction value to the value calculated by the LQI. This would take into account asymmetry and other factors. The LQI graph from the radio data sheet (fig. 4.2) was used to separate the LQI into different regions, each representing an ETX value. Some overhead was taken when calculating the ETX values for the given LQI. We had experienced a lower PRR for given LQI values than the data sheet, we had also seen that the LQI could be asymmetric. This led to the choice of "safe" ETX values, which probably underestimated the calculated ETX, but would help ensure links to be at least as good as the ETX indicated. The initially chosen values are shown in table 5.2. Comparing these numbers against figure 5.2 shows that these are very safe values. These are values are being reconsidered later in this document.

The LQI has proven to vary very much within short periods of time, and should therefore be averaged. An exponential-weighted moving average (EWMA) would therefore be used when updating the LQI values associated with a node when updating. This is calculated using the formula:

$$LQI_i = \alpha * LQI_{new} + (1 - \alpha) * LQI_{i-1}$$

where α is the weighing factor that is used, and LQI_{i-1} is the previous average. EWMA is an average that requires little memory, and is easy to implement. It is used by both TinyRPL and ContikiRPL. α was chosen to be 0.15, as the LQI varies pretty quickly. TinyRPL uses EWMA for ETX with $\alpha = 0.5$ for ETX, while ContikiRPL uses $\alpha = 0.2$.

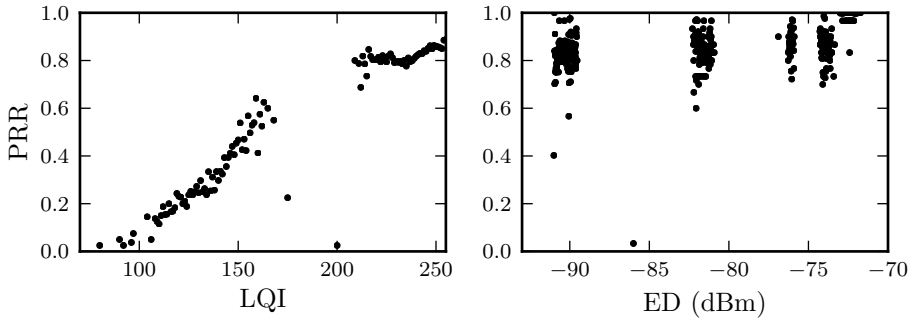


Figure 5.2: PRR vs LQI and ED from the hallway test

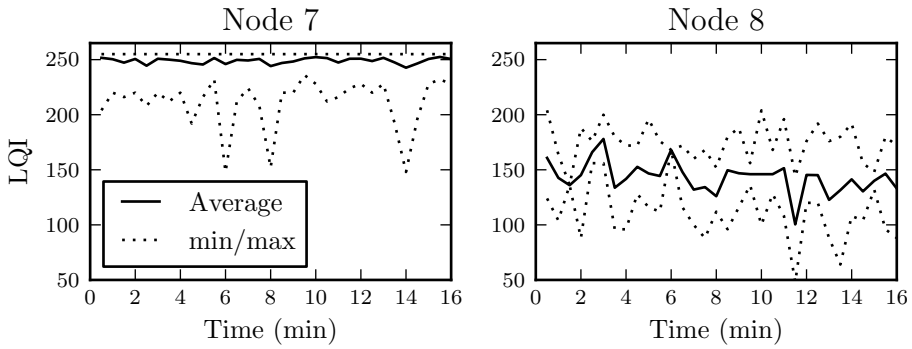


Figure 5.3: Variances in LQI for a short time period

This figure shows the average, minimum and maximum LQI values for two nodes. The values are averaged over 30 seconds, where a packet was transmitted every second.

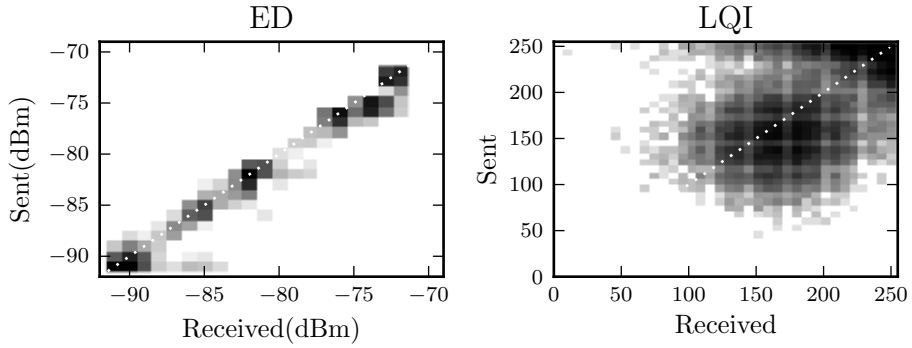


Figure 5.4: Link asymmetry

These graphs show sent vs received ED, and sent vs received LQI. A symmetric link would lay on the identity line (white and dashed). Color intensity increases with the number of observations. Note that color intensity is logarithmic. Small number of observations might therefore appear larger than they are.

LQI	ETX
>240	1
>220	2
>180	3
>160	4
>100	5
> 70	7
> 0	21

Table 5.2: Chosen ETX values for different LQI regions

In the case of routes with equal LQI, channel energy was to be used as a tie-breaker.

5.3 Choosing an Objective Function

As stated earlier, there exist two standardized OFs, OF0 and MRHOF. It was wanted to implement one of these, as a standard often is well tested and tried.

OF0 is a very simple OF, it does not offer any optimization, and does not guarantee any routes. It is also not designed with a specific metric in mind. It uses the rank_increase to classify link quality, and supports the choice of a backup feasible successor in addition to the preferred parent.

MRHOF includes some functionality that makes it more advanced than OF0. MRHOF can only be used for additive metrics. This requirement is met by ETX, which is the default path metric in MRHOF. It involves a hysteresis function. The hysteresis function makes the node stick with a parent unless the better route exceeds a given threshold. Also, MRHOF supports a parent set of variable size. MRHOF also has a parameter that enables us to filter out bad links.

The extended functionality in MRHOF made MRHOF the OF of choice.

Chapter 6

TinyOS

TinyOS[23] is an operating system (OS) targeted for WSNs and embedded devices. It is not what many users today would associate with an OS, as it doesn't offer any distinct user interface. It is a software component-based operating system, working like a framework to develop application specific code. TinyOS started in 2000 as a set of Perl script used by a handful of computer science researchers. Version 1.0 was released in 2002. TinyOS have evolved a lot since then. Averaging 25 000 downloads every year around 2012, it is used both commercially (e.g., Zolertia and Cisco's smart grid system), and for research. Some companies have their own diverged branches of TinyOS where own development is ongoing. TinyOS is as of this writing at version 2.1.2, which was released in August 2012.

TinyOS' is created for WSNs and embedded systems. TinyOS seeks to save memory by minimizing state information, minimizing computation and saving energy, and saving space by minimizing code size. This is done at the cost of reducing runtime flexibility and generality. The reason for these minimizations is the high cost for resources in embedded systems, and reduced memory usage.

Since WSNs are very difficult to debug when deployed, TinyOS also aims at making it hard to create bugs. Following TinyOS' evolution, stories about unsuccessful deployment of WSN networks due to bug and errors have been continuously reduced. One of the techniques that greatly reduce number of bugs is doing as much at compile time, rather then runtime. [24]

nesC, a programming language based on C, was created alongside TinyOS, and has evolved hand-in-hand with TinyOS. This is the programming language used for creating TinyOS applications.

Many OSs allows different application to run on top of the OS. In TinyOS the application and OS specific are compiled into *one* image, which is deployed on the microcontroller. This image, or TinyOS application, assumes control over all hardware, only one application can therefore be used at once.

6.1 TinyOS Applications

A TinyOS application is made of software-components [25]. The components used in TinyOS are called *configurations* and *modules*. The possible interaction between components is defined by *interfaces*. Component can both use and provide interfaces. A user of a given interface, can interact with a provider of the

```
interface Read<val_t> {
 command error_t read();
 event void readDone(error_t err, val_t val);
}
```

Listing 6.1: Example of TinyOS interface.

same interface. A technique called wiring decides which components are actually interacting together in an application. A user of an interface has to be "wired" to a provider of the same interface. Each component and interface is defined in its own file with the extension ".nc".

6.1.1 Interfaces

Interfaces define the possible relationships and interaction between components in TinyOS. It connects the name scopes of components together, making interaction between components possible. Interfaces are bidirectional, they consist of *commands* and *events*, each responsible for communication in a separate direction. The interface file does not define or implement any logic, it is merely a declaration of method names, their inputs, and return values. The actual implementation is done in modules. Listing 6.1 shows the interface definition for a common TinyOS interface, `Read`

Read is a generic interface. Generic interfaces are type-free, the implementation specifies which types should be used, and these are given as arguments. The types have to be given by both providers and users of the interface. Generic interfaces with different type-arguments are not compatible. The `Read` interface is used to read sensor information from sensors. Since Read is a generic interface, it can be used with sensors returning different types. e.g., it is compatible with sensors returning 8-, or 16-bit values.

The Read interface consists of one command, `read()`. `read()` does not have any input arguments, and returns an `error_t`[1]. The interface also defines the event `readDone()`[2], which has two input arguments, `err` of type `error_t`, and `val`, of type `val_t`. `readDone()` returns a void. Commands and events are explained in section 6.1.5.

6.1.2 Components

A component is either a module or a configuration. Modules and configurations are similar in the fact that they can both provide or use interfaces, however they have different purposes: Modules implements logic and functionality at the lowest level, and is the basic building brick in TinyOS. Configurations instantiate and "wire" components, but does not define any other logic than this.

[1] error_t is a TinyOS enum representing error messages, examples of error_t values are SUCCESS and FAIL.

[2] Split-phase operations like read() and readDone() are often used in TinyOS, they are discussed in section 6.3.1.

```
module moduleName {
 //Signature block
}
implementation {
 //Implementation block
}
```

Listing 6.2: Signature and Implementation block in TinyOS component.

A component declaration consists of two blocks, a signature block, and an implementation block (see listing 6.2). The signature block defines what interfaces the component uses or provides, while the implementation block is made of the actual implementation using nesC code.

Modules

A module is the the lowest-level component of a TinyOS application, this is where the logic is implemented. A TinyOS application consist of multiple modules, often separated by functionality or responsibility. A module store states in variables, and executable logic in methods. It shares many similarities with classes in other programming languages.

The logic in a module consists of nesC code, which at this level is almost identical to c code. All parts of a module is private. The allowed interaction with other components is defined by the interfaces the component use or provide.

Modules are by default singletons in TinyOS (like singleton classes, only one instance of an object exists in memory, even if it is instantiated more than once). A module can be defined as a generic module, which makes it a non-singleton object. Generic-modules can be instantiated more than once (examples of generic system modules are timers and queues).

For a module to interact with other components, it has to either *use* or *provide* interfaces.

An example of a module file can be seen in listing 6.3. This example shows us the file, which defines a module with the name moduleName, it uses the Read interface shown in listing 6.1. The `as` keyword assigns a different name to the interface, this improves code readability and makes it possible to the same interface multiple times. The code will from now on refer to the assigned name rather than original interface name. The implementation block shows us how the namespace in the interface is used. The command `read()` is *called* in line 10. In line 13 the `readDone()` event is implemented, this is done as a function. This example also shows the direction of commands and events. A user of an interface can call a command, which executes a method declared by the provider (like a normal function). Events go the opposite way, an event is signaled by the provider of an interface, this leads to execution of a method declared by the user of an interface.

```
1  module moduleName {
2   uses interface Read<uint16_t> as Temperature;
3   uses interface Read<uint16_t> as Humidity;
4   provides interface ABC;
5  }
6  implementation {
7   uint16_t temperature;
8
9   void increment() {
10   call Temperature.read();
11  }
12
13  event void Temperature.readDone(error_t result,
        uint16_t val) {
14   temperature = val;
15  }
16    ⋮
17 }
```

Listing 6.3: Example of module file.

Configurations

Configurations decide which modules are used, and how they interact. This is done by instantiating and wiring components. A configuration is different from a module in that it does not contain any code logic code. It only instantiates components, and decides which components interact with each other by wiring them. Like modules, configurations can provide and use interfaces. The signature block in a configuration definition is equal to a module file, but the implementation block differs. A configuration implementation instantiates and wire components. An example of a configuration is shown in listing 6.4. In the implementation block two components are instantiated, moduleName and TemperatureSensor [3]. The TemperatureSensor would be a component for a temperature sensor, and provide the Read interface.

6.1.3 Wiring

Wiring connects components that use and provide a given interface. This enables them to interact with each other.

Only interfaces that are of the same type can be wired. A wiring is done with a statement consisting of an "arrow" (-> or <-) with a module and interface name on each side. The "arrow" is pointing from the user of the interface to the provider of the interface. The arrow pointing towards right is the most common option. In this case the arrow can be read as "uses". An example of a wiring is shown on line 8 in listing 6.4. This shows a wiring between the components `moduleName` to

[3] Generic components are instantiated using "components new genericModuleName(arguments)"

```
configuration configurationName {
 uses interface interfaceA;
 provides interface interfaceB;
}
implementation{
 components moduleName;
 components TemperatureSensor;
 moduleName.Temperature -> TemperatureSensor.Read;
}
```

Listing 6.4: Example of TinyOS configuration.

```
module User {
 uses interface Read<type> as Temperature;
 uses interface Read<type> as Humidity
}

module Provider {
 provides interface Read<type>;
}

configuration{...} implementation{
 components User;
 components Provider;
 moduleName.Temperature -> TemperatureSensor;
}
```

Listing 6.5: Shortcut wiring.

`TemperatureSensor`. `moduleName` is a user of the interface `Read`, which is renamed to `Temperature`, and `TemperatureSensor` is a provider of this interface. If a component uses or provides an interface that is renames using the keyword `as`, the new named should be used in the wiring.

Shortcut wiring is very common TinyOS configurations. If a component uses or provides only one instance of an interface type, its name can be elided in the wiring statement. In listing lst:shortcut-wiring an example of this is shown. Since the Provider only provides one instance of the interface type used, the interface name is elided. The User however uses two interfaces of the same type, and the interface name must therefore be included.

6.1.4 Packages, Public and private components

Packages are collections of components working together with a common purpose. A package is a directory in the file-structure. The components which files reside in the package directory is part of that package.

Components can be either private or public. Private components have names that are suffixed with P, while public component names are suffixed with C. There are no other differences between a private and a public component. Private and public components are equal to the compiler. The distinction is merely a coding convention helping programmers to choose components to use.

According to the programming hints in the TinyOS programming manual [25], public components should be usable abstractions by themselves. If a component is not usable by itself, but part of a larger abstraction, it should be a private component. One should never wire to private components that is not a part of the same package.

6.1.5 Events and commands

Interfaces defined *commands* and *events*. Interfaces are bidirectional, and commands and events specify methods to be used in the different directions. Commands are methods specified by the provider and initiated by the user of an interface. Events are defined by the user and initiated by the provider of the interface.

Listing 6.3 shows us how the interface implementation is done for a user of the interface. Commands can be called from the user of the interface with the keyword `call` (see listing 6.3, line 10). The method name is prepended by the interface name, separated with a dot (.), similar to class methods in c++. Commands are declared in the component that provides the interface. The provider of an interface can send a signal, this is done with the keyword `signal` as shown in listing 6.6. A `signal` leads to the execution of an event. The event is declared by the user of the interface (see 6.3, line 13).

It is possible for multiple components to be linked to the same component simultaneously. These situations are normal in other languages where a class is a singleton, its method can be called from multiple other classes, this situation is called a fan-in. The users of an interface in TinyOS create a fan-in against commands. Since the interfaces in TinyOS are bidirectional, we can also get a situation where several users of an interface are wired to the same component. When the provider component signals an event, multiple user components of the interface will receive a signal and trigger the event. This is called a fan-out. In the case of multiple wirings, one call might lead to several methods being executed. Since this might lead to different return values, TinyOS uses combine methods. For instance is the default combine function for error_t to return `SUCCESS` if all return values are successful, if one or more return values are not `SUCCESS`, the return value will be `FAIL`.

6.2 Graphic representation

The Unified Modeling Language (UML) is the standard modeling language used for object-oriented programming. It is used to create visual representations of software systems. Amongst others, it defines ways to represent sequence and component diagrams. It does not define representation of component and interface used by TinyOS . The diagrams used in this paper are similar to the diagrams used

```
module moduleName{
 provides interface Read<uint16_t>;
 ...
}
implementation {
 uint16_t value;

 void someMethod() {
   signal Read.readDone(SUCCESS, value);
 }
}
```

Listing 6.6: Provider of interface.

by nesdoc, also described in [26]. These are shown in figure 6.1.

6.3 nesC

nesC is the programming language used to make TinyOS applications. Some of its syntax and use is very similar to c, but there are some differences. Dynamic memory management is for instance strongly discouraged in TinyOS. nesC also introduced some new concepts not present in c at all. The biggest noticeable difference is the introduction of software components, this being interfaces, configurations, modules and wiring.

nesC uses more explicit variable types than normal in other languages. Instead of using an `int` or `long`, one uses variables that specify the size and type, e.g., `uint_8t`, which specifies an unsigned integer of 8 bits. `int` and `long` data types are platform dependent, and their size can vary from platform to platform. The types used in TinyOS are very specific, and their size is equal on all platforms.

6.3.1 Split-phase operations

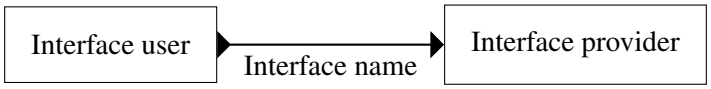
In TinyOS every long-running operation is split-phase operation. As most hardware operations are split-phase, this creates flexibility where hardware components can be easily interchanged with software components if the necessary hardware is not present.

In blocking systems operations does not return before they are finished. While the operation is running, the rest of the program is blocked (waiting for the operation to finish). Computers usually solve this problem by using threads that are running simultaneously. This solution is not very good for embedded systems like WSNs, as it requires a good amount of RAM. TinyOS uses hardware interrupts and tasks to perform background computations, these will be discussed in section 6.3.2.

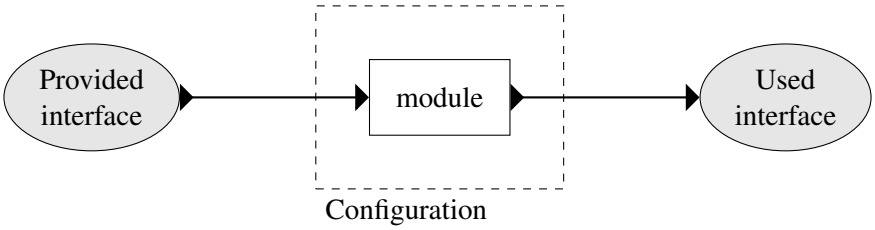
Split-phase operations are non-blocking and asynchronous. They are operations are bidirectional. A split-phase operation is initiated by calling a method, which returns immediately. When the operation that was initiated is completed, a

(a) TinyOS components are represented by rectangles. Configurations are made of a double line, where as modules are made by a single line. The component name is written inside the rectangle.

(b) Generic components are rectangles like normal components, but use a dashed line instead of a solid line.



(c) Component wirings are represented by a line with a triangle in each end. The triangle points toward the component that provide the interface, and out of the component that use the interface. The interface name used to wire the components are written by the line.



(d) A configuration can provide and use interface externally. The diagram for the inner components of a configuration represents these interfaces as ellipses surrounding the interface names. The wiring is represented like between normal components. A dashed outline surrounds the components that are used in the configuration

Figure 6.1: "TinyOS components are represented by rectangles. Modules use a single line, whereas configurations use a double line"

```
void startOperation() {
 temperature = call
    Read.read();
}
```

(a) Blocking

```
void startOperation() {
 Read.read();
}

event void Read.
    readDone(...) {
 temperature = val;
}
```

(b) Split-phase

Listing 6.7: Example of blocking and split-phase operations

```
task void taskName() {
  i=5+5;
}
```

Listing 6.8: Declaration of a task.

callback is issued. In TinyOS split-phase operations are initiated with a command, and the callback operation is accomplished by signalling an event. Listing 6.7b shows usage of the split-phase interface `Read` interface (shown in 6.1 on page 38).

6.3.2 Execution Model: Tasks,stack and the scheduler

TinyOS execution model is based on split-phase operations, tasks and interrupt handlers [27].

All code in TinyOS applications are running from either tasks or interrupt handlers. A task is like a very lightweight procedure call. It does not have any arguments or return values. While a task has to be posted to run, interrupts are asynchronous and can happen at any time, and also interrupt other running code. TinyOS distinguishes between code that can be run only from task (synchronous) and code that can be run from interrupt handlers (asynchronous). Writing async code is more challenging as one have to deal with concurrency of variables. Writing async code is not often needed when making many TinyOS applications, and will not be described.

Tasks are used in TinyOS to schedule code to run at a later time. All the code executed by a task is synchronous. Tasks are defined and declared in modules. A task declaration is shown in listing 6.8.

Tasks do not have a return value, or any input arguments. The code in a task can do everything a normal module can do; use variables, send signals, call commands and execute methods. A task is put on the stack using the keyword `post` as shown in listing 6.9. Since a task does not have any return value or input arguments, the stack can be very simple and requires little memory. All state is saved in the module.

A task can exist only once on the stack. Trying to post a task that already is on the stack, will return a `FAIL`. Posting a task always return `SUCCESS` otherwise.

```
void someMethod() {
  post taskName();
}
```

Listing 6.9: Posting a task.

```
interface Init {
  command error_t init();
}
```

Listing 6.10: Init interface.

Tasks are run to completion, one-by-one, and cannot be preempted by any other tasks. Since tasks are run to completion before the next task can be executed, tasks containing long-running operations can limit system-responsiveness. It is therefore recommended to split long-running operations into multiple tasks. This can be done by creating multiple tasks for the long running operation, or be keeping a state variable, and have the task repost itself.

Tasks make it easy to create split-phase operations in software. The initiating command in the split-phase call posts a task to perform the needed operation, and returns immediately after doing this. When the task is running and the operation finishes, it signals the proper event for this.

6.3.3 Boot sequence

TinyOS applications does not have a main() method like many other programming languages. The boot sequence is defined by a number of interfaces and components, it contains of four steps.

1. Scheduler initialization
2. Component initialization
3. Signal that the boot process has completed
4. Run the scheduler

The application-level component in control of these steps is `MainC`. `MainC` only provides the `Boot` interface, and uses the interface `Init as SoftwareInit`. The `Init` interface is shown in listing 6.10. The real work is done by the module RealMainP. RealMainP use two additional interfaces, `Init as PlatformInit` and `Scheduler`. Since RealMainP is suffixed with a P, it is a private component, which we should never wire against.

The first step in the boot process is to initialize the scheduler, this means that the later initialization processes are able to post tasks. Tasks are run after each of the specified steps.

After the scheduler has been initialized, the platform is initialized. A platform in TinyOS is a hardware platform, which is a specific type of node. A platform is made up by several components. The platforms are defined in the folder platforms/ in the TinyOS directory, examples of supported platforms are micaz, telosb, and iris, (the latter will be presented later). Different platforms often use some of the same chips (e.g., microcontroller, radio), which are defined in the chips/ folder in

```
interface Boot {
  event void booted();
}
```

Listing 6.11: Boot interface.

the TinyOS directory. The platform wires chips together. This makes it possible to share implementation of chips between different radios. Each platform has a PlatformC component, which is automatically wired to the `PlatformInit` interface of RealMainP.

Software initialization is done using the `SoftwareInit` interface from `MainC`. A component that needs to initialize some state when booting the application can use the `Init` interface and wire it to `MainC`. The MainC will call the command init(). Since multiple components are wired to `SoftwareInit`, this will create a fan-out, and all the components that provides an `Init` interface that is wired to MainC, will be initialized.

When all the initialization has been done, `MainC` uses the Boot interface (listing 6.11 to signal that the boot process is finished and the application is ready to start. This is done through the event `booted()`

The event `booted()` is the equivalent to the main() function found in many other programming languages. For a component to execute at boot, it has to use the `Boot` interface, and wire it against `MainC`. When the initialization is finished and the `booted()` signal is sent, the code will be executed.

6.4 Hardware abstractions

TinyOS uses a Hardware Abstraction Architecture (HAA) consisting of three layers. Hardware abstractions abstract hardware specific code. This makes it possible and easy to write hardware-independent applications. Hardware abstractions can however contain large simplifications, removing the possibility to use hardware specific properties, which might be wanted. The three layers of abstractions solve this problem. Using TinyOS it is possible to create hardware independent applications using hardware abstractions, but also possible to access hardware-specific properties by using deeper abstractions. Using more hardware-specific properties is done at the cost of removing hardware independency. This gives the developer the choice between hardware independency or hardware-specific properties. Later in this thesis we will see how we have to use hardware-specific properties of the radio to get link-quality information, which in turn makes the application hardware-dependent. The abstraction model used in TinyOS is described in TEP2 [28].

6.4.1 Hardware Abstraction Architecture

The Hardware Abstraction Architecture (HAA) has 3 distinct layers. They offer different degrees of hardware abstraction. The three layers are:

- Hardware Presentation Layer (HPL)
- Hardware Adaption Layer (HAL)

- Hardware Interface Layer (HIL)

Hardware Presentation Layer is the lowest level layer in the HAA. It is positioned right above the HW/SW interface. Access to the hardware is made using memory or port mapped I/O, and the hardware can request servicing by signaling and interrupt. HPL presents the hardware intricacies and exports a more readable interface. Each HPL interface is determined by the specific hardware it represents. Hardware that are similar should however have a similar structure. HPL components do not contain any state.

Hardware Adaption Layer is above HPL in the HAA structure. It uses the raw interfaces provided by HPL. The HPL contains most of the implementation, and is the core of the architecture. Some state is allowed to be saved in the HPL. HAL abstractions are made for concrete device classes and platform. They provide rich, customized interfaces, making them easy to use and understand.

Hardware Interface Layer is the highest-level abstraction interface. It creates hardware-independent interfaces from the hardware-specific HPL interfaces. These standard interfaces provide access to functionality typically presented by hardware of the same functionality. This makes HIL interfaces less specific than HAL interfaces, and might remove some functionality at the cost of independency. HIL interfaces hide all hardware-specific code, and is the ones used when creating hardware-independent applications.

The HPL is rarely used when developing TinyOS applications. To create hardware-independent application the HIL is used. More specific hardware functionality is accessed through HAL interfaces. Using HAL interfaces makes the application hardware-dependent, as HAL interfaces differ for hardware components. This is shown in figure 6.2

6.5 Communication

Communication is a vital part of Wireless Sensor Networks. Two communication types are normal in TinySO. Communication between nodes using a wireless communication over 802.15.4, and communication between a node and a computer using over a serial connection. Radio and serial communication in TinyOS are very similar, they use the same message buffer (`message_t`), and the same interfaces. Crossing the boundary between nodes and computers is therefore very easy. In addition to this TinyOS uses Active Messages (AM) to multiplex access to the radio, making it easier to for multiple services in an application to share the communication resource. AM is TinyOS specific, and is not interoperable with other operating systems.

TinyOS also supports IPv6 over Low power Wireless Personal Area Networks (6lowpan), which allows IPv6 packets to be sent and received over 802.15.4 networks. This enables communication between different operating systems.

While TinyOS have some interfaces providing multi-hop communication, these are tied together with routing protocols, and will not be discussed.

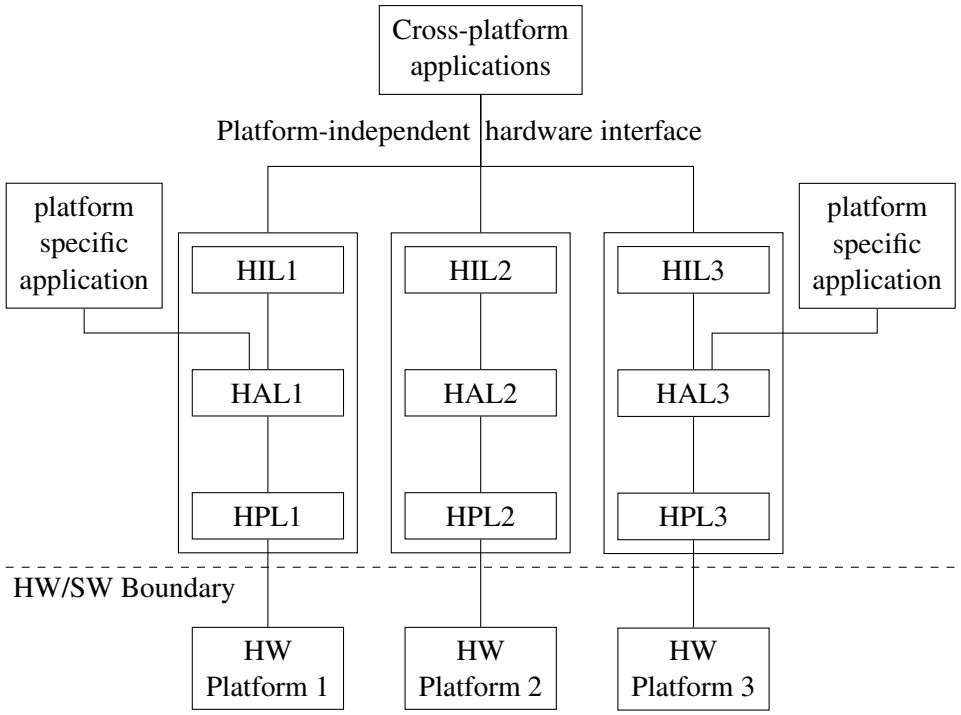


Figure 6.2: Hardware Abstraction Architecture in TinyOS

802.15.4 Header	AM type	data	802.15.4 CRC

(a) TinyOS Frame (T-Frame)

802.15.4 Header	6lowpan	AM type	data	802.15.4 CRC

(b) Interopable Frame(I-Frame)

Figure 6.3: Frame types used by TinyOS for communication.

6.5.1 Active Message Layer

TinyOS uses a layered model for its communication stack. Each layer has its own header and footer surrounding the payload. All of this is inserted into the payload of the lower layer, which in turn has its own footer and header.

Active Message (AM) is the lowest network layer exposed by TinyOS.It implements unreliable single-hop communication. AM messages contain a 8-bit type number. The type number is used to identify and packet types and dispatch them. This makes it easy to separate radio traffic into different services in components. The AM layer is often implemented right on top of the radio. AM uses `message_t`, described in section 6.5.4, as its message buffer. TinyOS AM messages are transmitted over 802.15.4 using the TinyOS-frame (T-frame) shown in figure 6.3. This format is for networks where there are no other operating systems using the same communication channel [29]. The I-frame is the default frame used by TinyOS.

AM communication uses several interfaces for basic communication, some of these are:

```
interface AMSend {
  command error_t send(am_addr_t addr, message_t* msg,
      uint8_t len);
  command error_t cancel(message_t* msg);
  event void sendDone(message_t* msg, error_t error);
  command uint8_t maxPayloadLength();
  command void* getPayload(message_t* msg, uint8_t len
      );
}
```

Listing 6.12: The AMSend interface

- *AMSend* is a split-phase interface used to fill and send packets. It is shown in listing 6.12
- *Receive*: an interface for receiving packets. It is shown in listing 6.13
- *AMPacket* has accessors for most `message_t` abstract data types (e.g., AM source and destination addresses and AM packet types).
- *Packet*: has access methods for the payload and its length. The Packet interface is used to read or write the payload.

[30]

6.5.2 Basic sending and receiving

AMSend

AMSend is the interface used to send and receive AM packets. Its specification is shown in listing 6.12. AMSend is a split-phase interface. It has a queue of depth 1.

Packets are sent using the command `send(...)`, its input arguments is a AM destination address(`addr`), and a `message_t` buffer being the actual package. The `len` argument specifies the size of the data in the packet payload. A successful transmission of a packet is signaled by the event `sendDone(...)`.

Since the queue depth is only one and the message buffer is passed by reference to the `send` method, the user must be careful when using this interface. In the time after the `send(...)` command have been called, and the `sendDone(...)` event has been signaled, the message buffer must not be changed at all, as this might affect the process of sending the message.

Receive

The `Receive` interface is used when receiving messages from the AM layer, its definition is shown in listing 6.13. The receive interface has a *buffer-swap* policy. Using this interface wrong is one of the most common ways of introducing bugs in TinyOS applications[31]. The buffer-swap policy ensures that data-rate mismatches do not occur between layers. The buffer-swap policy states that the handler of the event must return a message buffer that it will not use later, this can be done by:

```
interface Receive {
  event message_t* receive(message_t* msg, void*
     payload, uint8_t len);
}
```

Listing 6.13: The Receive interface

```
generic configuration
   AMSenderC(am_id_t
   AMId) {
  provides {
    interface AMSend;
    interface Packet;
    interface AMPacket;
    interface {Packet-
       Acknowledgements}
        as Acks;
  }
}
```

(a) AMSenderC

```
generic configuration
   AMReceiverC(am_id_t t
   ) {
  provides{
    interface Receive;
    interface Packet;
    interface AMPacket;
  }
}
```

(b) AMReceiverC

Listing 6.14: AMSenderC and AMReceiverC configurations

- returning the same buffer that is used in the `sendDone(...)` event. The buffer *must not* be accessed later, if any data is needed from it, this should be copied in the handler.
- swap the buffer and return a different buffer.

The buffer returned in the handler is used by the lower layer to store the incoming packets, if this does not exist, there is no room to save the received packets.

The basic components used to send and receive are *AMSenderC* and *AMReceiverC*. Their signatures are shown in listing 6.14. Both These are generic components and has to be instantiated with the keyword `new`(e.g, `new AMReceiverC (AMType))` keyword. One component has to be instantiated for each AM type to be used, the AM type is given as an input argument to the The AM type to be used for that component is given by the input argument when instantiating the component. For each AM type to be used, separate objects have to be instantiated.

6.5.3 Snooping messages

In addition to the components for basic sending and receiving, two components exist to snoop messages, `AMSnooperC` and `AMSnoopingReceiverC` [30] . Snooping messages is the process of receiving and reading messages that are not intended for us. `AMSnooperC` and `AMSnoppingReceiverC` have the same signature as `AMReceiverC`. `AMSnooperC` is used to receive

```
typedef nx_struct message_t {
  nx_uint8_t header[sizeof(message_header_t)];
  nx_uint8_t data[TOSH_DATA_LENGTH];
  nx_uint8_t footer[sizeof(message_footer_t)];
  nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

Listing 6.15: message_t the TinyOS message buffer.

messages that are not destined to the nodes address, or the broadcast address. `AMSnoopingReceiverC` receives all messages, regardless of their destination address. Because of the buffer-swap policy, an `AMSnoopingReceiver` *cannot* coexist with a `AMReceiverC` or `AMSnooperC` for the same AM type.

6.5.4 message_t

`message_t` is the message buffer type used in TinyOS. It is used for both serial and radio communication. Its structure can be seen in 6.15.

From the listing we can see that message_t is a `nx_struct`. The `nx_` specifies an external type. External types are an extension to C. It allows definition of types with platform independent representation. TinyOS uses `nx_` types for all communication. There is defined nx types for structs(`nx_struct`), unions(`nx_union`), and integers(`nx_uintN_t, nx_intN_t` and more for N=8,16,32,64). External structs and unions can only contain external types. [32]

Like many others network messages, message_t consists of a *header*, a *data*, a *footer*, and a *metadata* field. The size of the data payload is defined in `TOSH_DATA_LENGTH`, which is 28 bytes by default. `TOSH_DATA_LENGTH` can be redefined at compile time, but is constant for all instances in an application. Received packets with data-fields larger than `TOSH_DATA_LENGTH` will be discarded. The fields in `message_t` objects should not be accessed directly, only by mutator functions. The header and footer contain elements which are normal in other network message structures. The metadata field contains options from the radio. For the RF230 radio, this information includes information about link strength and quality.

The actual structure of the header, footer and metadata is link layer dependent. Link layers have different header and footer structures, these must therefore be specified by each separate radio implementation. `message_header_t`, `message_footer_t`, and `message_metadata_t` also has to be specified for each platform. The platform definition should include all the different link layers present on the platform in a `union`. Since the size of a union is the size of the biggest element in it, this ensures each element is big enough for all the different link layers, and that the same buffer object can be used when crossing different link layers [33]. The platform file for the IRIS platform, which will be presented later is shown in listing 6.16.

The IRIS platform has a rf230 radio, and a serial stack, which we can see are both included in the message_header. This makes it possible to use the same `message_t` objects on the rf230-radio, and the serial stack, messages can easily

```
typedef union message_header {
 rf230packet_header_t rf230;
 serial_header_t serial;
} message_header_t;

typedef union message_footer {
 rf230packet_footer_t rf230;
} message_footer_t;

typedef union message_metadata {
 rf230packet_metadata_t rf230;
} message_metadata_t;
```

Listing 6.16: Platform message definitions for the iris platform

cross link layers [30] .

6.5.5 ActiveMessageC

ActiveMessageC is the HAL component providing access to AM components. ActiveMessageC is used by AM components previously mentioned. ActiveMessageC provides the same interfaces as these other components, and could be used instead of these. However the AMSenderC provides a virtualization of the radio with 1-deep queue for each instantiation of the component. Users of the component do therefore not need to know whether the radio might by occupied by any other users.

ActiveMessageC must however be used. As the radio is not turned on by default, the interface `SplitControl` provided by ActiveMessageC has to be used to turn on the Radio.

There exist ActiveMessageC HIL components for each radio. The HIL components have a prefix to them (e.g., RF230ActiveMessageC for the RF230 radio). To gain access to some metadata fields, these must be used instead of the HAL ActiveMessageC components.

6.5.6 Serial communication

Serial Communication in TinyOS is very similar to radio communication. They both use the AM layer, and both uses the same message buffer. This makes it possible to easily receive messages over the radio and forward them to the serial port.

The components used for serial communication is also very similar to the ones used for radio communication. ActiveMessageC, AMSenderC, and AMReceiverC is simply prefixed with Serial (e.g., SerialActiveMessageC). They are so similar that replacing the components and rewiring is the necessary action to communicate over the serial port instead of the radio.

TinyOS provides several options from communicating with a TinyOS node from the computer using a serial connection. There exists software to:

- Print raw information from received messages.

- Print debug messages received by using dbg(...) functions provided by TinyOS.
- Create a TCP socket forwarding messages using the serialforwarder.

SerialForwarder is an application implemented in both c and c++. It creates a listening tcp socket at a tcp port for a given connection. The tcp socket is used to forward messages between the socket and the serial port. SerialForwarder multiplexes connections. This makes for multiple clients to use the tcp connection against the mote simultaneously. TinyOS comes with a debug library that makes debugging over the serial connection easy. A library and component is used in the TinyOS code. The `debug(...)` function sends text messages over the serial port to the computer. A java application included with TinyOS is used to read debug messages from the serial port. It is not possible for two applications to have the serial port open simultaneously. Using the serialport it is easy to use the debug application while running a separate application also using the serial port. It is therefore easier to create applications utilizing the serial port, separate applications can be created with separate responsibilities. A separate application can use the serial connection for data gathering, while the debug application simultaneously prints debug messages.

For some programming languages TinyOS provides APIs that makes it possible to interact directly with motes connected to a serial port (e.g., java, c and c++). TinyOS provides APIs to communicate with motes through the serial forwarder for additional programming languages to the ones previously mentioned (e.g., python).

message interface generator (mig) is a tool included with the nesC compiler. mig creates code to process messages used in TinyOS. These messages can be user-defined. The message must be a `struct`, `nx_struct`, `union`, or `nx_union`. The arguments used with mig specify which message types code should be generated for, the source file that contains the definition for the message type. Arguments also decide which programming language the created code should be in, and necessary options (e.g., class name). The generated methods from mig remove the need to manually extract the necessary fields from messages. Changing the structures of messages also becomes much easier, although unwanted, this might happen in the development of applications.

6.6 BLIP 2.0 and TinyRPL

In addition to ActiveMessageC, TinyOS includes a IPv6/6lowpan implementation called Berkeley Low-Power IP stack. Due to the restraints in LLN network, 6lowpan was created to enable IPv6 communication between low power devices with "normal" using IPv6. IPv6 requires a minimum MTU of 1280 bytes [34], and the maximum size in 802.15.4 is 102 bytes, 6LoWPAN fragments packets to make these compatible [35]. 6LoWPAN specifies header and address compression to be able to use IPv6 128 bit addresses in 802.15.4. Using UDP with IPv6 would leave 33 bytes out of 102 to the application payload. This would introduce alot of overhead (= 68%) for the headers. Header and address compression is therefore done.

BLIP 2.0 is the de-facto IPv6 stack in TinyOS. It implements header compression, neighbor discovery and DHCPv6. BLIP 2.0 also uses a routing table,

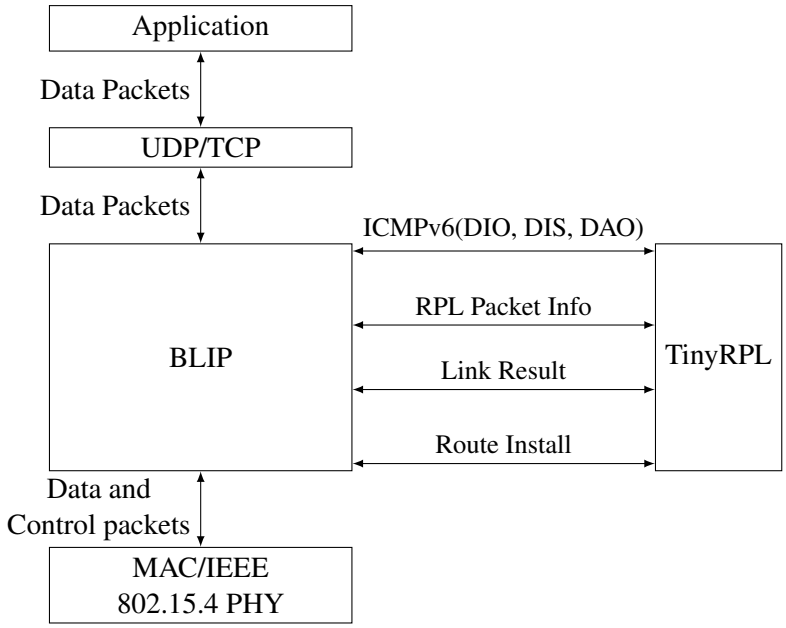


Figure 6.4: BLIP and TinyRPL network stack.

and routing functionality. The BLIP 2.0 implementation also supports UDP, which makes it possible to easily interact with normal computer applications. A network using BLIP 2.0 is usually a bit different from many other TinyOS applications. A normal TinyOS application often has a specific computer application that receives application specific packets from the network over the serial port. Using BLIP 2.0, this would look more like a traditional computer network. A router with an IPv6 and 6LoWPAN implementation connects the WSN to another network. This enables both networks to communicate. Messages between the WSN and the computer are for instance sent as UDP. Doing it this way strongly supports the thought of the "Internet of Things", where every node is reachable from the Internet. Included with TinyOS is a ppp application, which sets up a tunnel between a computer and a serial node. This makes it easy to connect networks using BLIP 2.0 with normal networks.

6.6.1 TinyRPL

TinyRPL is a RPL implementation in TinyOS. It is based on the most recent version of RPL draft 17. It supports the drafts basic options, creates upward and downward routes, support storing mode of operation. It can use OF0 or MRHOF, with the ETX metric. TinyRPL strongly interacts with the BLIP implementation. The TinyRPL implementation have also been tested and found compatible with Contiki's RPL implementation [36].

An overview of the BLIP 2.0 and TinyRPL stack can be found in figure 6.4.

6.7 Power management

The consumed power of a node is built up by the power consumption of several components. Their power usage is dependent on whether they are on or off, and what they are doing. Nodes should consume as little power as possible, and be able to operate for a long time. It would be ideal if the operating system controlled the power state of the different components, optimizing it as much as possible. Many OSs only provide necessary power controlling functionality, leaving most of the responsibility to the specific application. TinyOS automatically controls power for many operations. Automatic power control in TinyOS comes very close to hand-tuned applications in many of cases [37].

Microcontroller

The power management of the microcontroller is done automatically by TinyOS. TinyOS automatically figures out the lowest possible power state the processor can be in at the given moment, and sets it to this state. Processors often have different levels of power states, and how deep they can go depend on what functionality is in use, and should be able to "wake" the microcontroller. A timer that is running, or an active USART, can stop the microcontroller from entering a deeper sleep state.

Peripherals and subsystems

Some peripherals and subsystems must be manually controlled by the application developer. Examples of this are the radio and USART connection. These can constantly consume a lot of power when enabled. They have to be explicitly enabled while the application is running in most situations. Interfaces such as these are controlled using the `StdControl`, `SplitControl`, or `ASyncStdControl` interfaces.

Parallel vs serial sensor reading

Other peripherals, like sensors, are automatically powered when needed. These peripherals are accessed on demand. Since they only need to be powered in short periods while they are being used (read the value of a sensor, write something to flash), the developer doesn't have to control these. There are however some techniques techniques that enable TinyOS to optimize energy management when doing tasks like these. One of these methods is to call methods that use peripherals in parallel instead of in serial. This can be applied when reading several sensors at the same time. The serial method of doing this would be to trigger the reading of a sensor when another sensor reading is finished. Doing this in parallel, calls all the read methods at the same time. Sensors are often connected to the same bus, part of the process of reading a sensor involved turning this bus on and off. If the sensors are being read serially, the bus will be turned off when one sensor is read, but shortly after turned on again to read the next sensor. Doing this work in parallel enables TinyOS to schedule these operations, this could involve turning the bus on once, read the sensors, and turn the bus off again. This would be more energy efficient, as the time the bus is powered is reduced. Listing 6.17 shows how to do read operations in parallel and serial [38].

```
void startOperation() {
 call Temperature.read();
}

event void Temperature.readDone(...) {
 //Some work
 call Humidity.read();
}

event void Humidity.readDone(...) {
 //Some work
}
```

(a) Serial

```
void startOperation() {
 call Temperature.read();
 call Humidity.read();
}

event void Temperature.readDone(...) {
 //Some work;
}
```

(b) Parallell

Listing 6.17: Example of reading sensors in serial and parallel.

6.8 Low Power Listening

The radio is one of the most power consuming components in a node. Power consumption while receiving is very close to power consumption while transmitting in many radios. There is not much the operating system can do to reduce energy consumption when transmitting.Many WSN applications have very little network traffic and nodes spend most of their time listening idle for network messages, without actually receiving any messages. Network activity in WSN scenarios can be very sparse, and the nodes are idle-listening most of the time. This leads to unnecessary high power consumption. Idle listening power consumption can be reduced by turning the radio off, and periodically turn it on to detect if someone is trying to transmit. This is called duty cycling [39] .

6.8.1 Media Access Control (MAC) protocols

Low power Media Access Control(MAC) layers for WSN is an area where a lot of research has been done. This has resulted in many different MAC protocols. Some popular low power MAC protocols for WSNs are S-MAC, B-MAC and X-MAC. The MAC layer is also, amongst other things, responsible for avoiding collisions, use the channel efficiently, and share the channel fairly.

The MAC layer can reduce the radios energy consumption by duty cycling the radio in periods of idle listening. The cost of this can be reduced throughput, and higher energy consumption for the transmitter.

When the radio is duty cycling it is often in the off state. The radio is periodically turned on to check if there is any traffic on air. Energy consumption is reduced when idle listening since radios energy consumption is several orders less turned off compared to when listening. Since a receiver's radio is not always on, the transmitter will have to transmit for a longer time to ensure the receivers radio wakes up, and receives the message. Three states often exist when receiving messages with low power. While the radio is turned off, the receiver is *idle*, the radio is briefly turned on to *detect* any transmitting nodes, if a transmission is detected, the radio stays on to *receive* the message.

1. The radio is turned off in an *idle* state to save energy. The time in this state is called the sleep period.
2. The radio is briefly turned on to *detect* if any active transmissions. If no transmissions are detected, the radio returns to idle mode.
3. If a transmission is detected, the radio enters a *receive* state. In this state the radio is listening like it normally would.

The process of detecting that a node is transmitting, affects both the receiver and the transmitter, as stated, transmitting nodes have to transmit more to make sure that the receiver is able to detect the transmission. This process is done differently in different MAC protocols. B-MAC and X-MAC will be briefly explained, these are both *single-layer protocols*, utilizing information from only one layer for the detection state, B-MAC uses physical-layer carrier sensing, while XMAC uses link-layer wakeup packets. The cross-layered protocols BoX-MAC-1 and BoX-MAC-2 will also be introduced.

B-MAC [40] is a CSMA MAC protocol designed for low power WSNs. B-MAC

utilizes layer 1 information to detect activity on the link. MAC uses Clear Channel Assessment (CCA) and packet backoffs for channel arbitration. B-MAC duty cycles the radio to reduce power consumption. The channels energy is periodically sampled to check for activity. CCA is normally used to check that a channel is not busy before transmitting. This is done by comparing the channel energy to the noise floor. B-MAC uses CCA the other way around, to check whether for active transmissions. If an active transmission is found, the radio enters the receive state. B-MAC prepends every transmitted message with a preamble. This preamble must be at least as long as the receivers sleep period. When a receiver finds that a transmission is active, it must keep its radio on long enough to receive the packet. If no packet is received within a certain time (e.g., if a false detection has been made), the radio is turned off by a timer. A B-MAC example is shown in figure 6.5a. When using B-MAC, all nodes detecting the preamble have to turn on their radio while the preamble is being sent. This overhearing is a waste of energy, as radios have to be listening to packet not destined for them. This introduces a lot of aggregate energy consumption. The long preamble also increases network latency, as it has to be transmitted for a long time, since the sender does not know whether the receiving node is listening or not.

X-MAC [41] tries to solve some of B-MACs problems using a shortened preamble approach. Like B-MAC, radio duty cycling is used to save energy. XMAC uses layer 2 information to detect if the channel is active. Instead of sending one large preamble like B-MAC, X-MAC sends strobed preambles (wake up packets). The preambles are short and contain destination information. Since the recipient only uses layer 2 information, its detection phase must be long enough to be able to read a whole packet. The destination information allows non-recipients to go right back to sleep. A pause is inserted between every transmitted preamble. When a node detects a transmission destined for it, it can send an early acknowledgment in the gap between two preambles. This informs the transmitter node that the receiver is listening, and the packet can be sent immediately right away. The unnecessary transmission of preamble after the receiver has woken up, is removed. This reduces time spent transmitting and receiving, which reduces energy consumption. Since less preamble have to be transmitted, network throughput and latency can also be improved. An example of X-MAC can be seen in figure 6.5b.

BoX-MAC [42] BoX-MAC-1 and BoX-MAC-2 are cross-layer MAC layers. BoX-MAC-1 uses mostly layer 1 information to detect activity, while BoX-MAC-2 uses mostly layer 2 information to detect activity. Both protocols get inspiration from B-MAC and X-MAC, as can be seen in their name. They can however use up to 50% less energy than X-MAC, and 30% less energy than B-MAC.

BoX-MAC-1 [42] BoX-MAC-1 is similar to B-MAC, but is improved and utilize layer 2 information to decide whether to listen or not. The preamble is made up by continuously transmitting packets, which contain destination information like X-MAC. However, the pause introduced between preamble packets in X-MAC is not used. Like B-MAC, the full length of the preamble is constant, and always

transmitted. The added layer 2 information prevents non-recipient from waking up unnecessary.

BoX-MAC-2 [42] BoX-MAC-2 is most similar to X-MAC, but additionally use layer 1 information to detect transmissions. BoX-MAC-2 first uses CCA like B-MAC to detect activity. This reduces the time the radio has to be in the detection state, as it does not have to read a whole message like X-MAC. Since the detection does not require reading a whole packet, longer messages can be sent transmitted[4]. The added time needed to read the whole packet is not an issue now. BoX-MAC-2 therefore transmits the original data-packet from the beginning, instead of the preamble. This also removes the need for an early acknowledgment message from the receiver to indicate that it is listening. Like XMAC, small pauses are inserted between the packets. When a node detects an active channel using CCA, it turns on the radio long enough to receive a whole packet sent by the transmitter. The receiving node can now return an ack when it has received the packet. This makes the transmitter stop. BoX-MAC-2 is shown in figure 6.5c.

6.8.2 Low Power Listening Interface

TinyOS provides the ability to use a low power MAC protocol, this is called Low Power Listening (LPL) in TinyOS. The first LPL implementation in TinyOS used the B-MAC protocol. BoX-MAC-2 is the currentLPL MAC protocol used by TinyOS[43]. TinyOS uses the `LowPowerListening` interface to control the low power functionality. It is provided by `ActiveMessageC`. LowPowerListening is currently supported on 3 radios (cc1000, cc2420 and rf230). As can be seen by the interface in listing 6.18a, one can set the local and remote wakeup interval. The wakeup interval is given in milliseconds. The local wakeup interval decides how often the radio should check the channel for activity. The Low Power Listening wakeup intervals can also be set in the applications Makefile, this is shown in listing 6.18b. The remote wakeup interval is the receivers wakeup interval, and is set for every packet sent, and is used to decide how long a packet has to be transmitted. This enables nodes in a network to have different wakeup intervals. In a heterogeneous network consisting of both powered and battery-operated nodes, the powered nodes can be always listening. Power consumption can thus be reduced when transmitting to power operated nodes.

6.9 TOSSIM

TOSSIM is a discrete event simulator for TinyOS WSNs. TinyOS applications can be compiled to the TinyOS frameword. This makes it possible to run TinyOS applications on computers, in repeatable and controllable environments. The task of debugging and verifying a TinyOS application becomes much easier when run in a simulated environment. TOSSIM simulates the entire TinyOS application. It works by replacing components with simulation implementations. There exist generalized components, but also components that simulate a specific chips behavior.

[4] In XMAX, the packets are short to reduce time spent detection

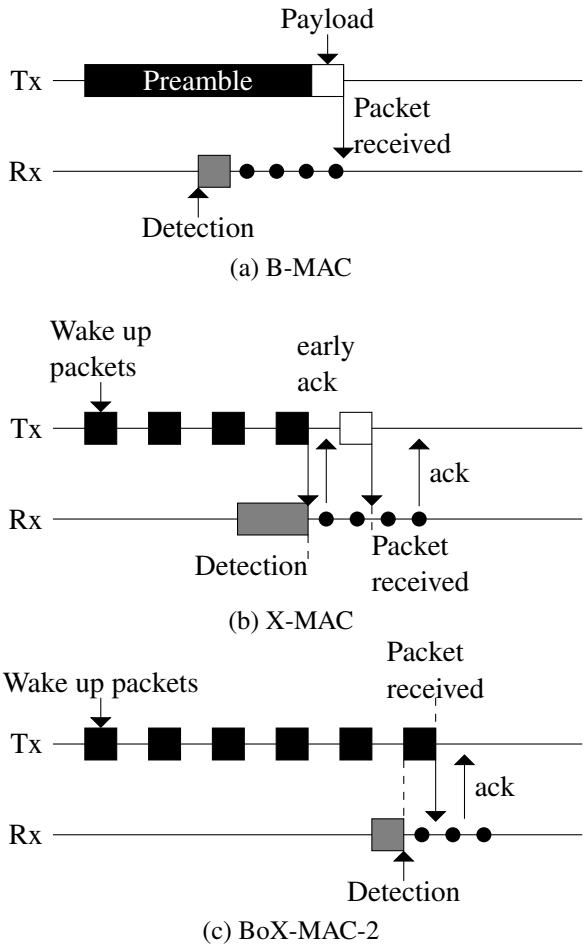


Figure 6.5: Low power saving MACs

This figure shows listening (tx) and transmitting (rx) node. A blank line indicates that the radio is in the off. Dots along the line indicate that the radio is listening. Grey boxes indicate the node is detecting if there is an active transmitter.

```
interface LowPowerListening {
  command void setLocalWakeupInterval(uint16_t
     intervalMs);
  command uint16_t getLocalWakeupInterval();
  command void setRemoteWakeupInterval(message_t *msg,
      uint16_t intervalMs);
  command uint16_t getRemoteWakeupInterval(message_t *
     msg);
}
```

(a) LowPowerListening interface

```
CFLAGS += -DLOW_POWER_LISTENING
CFLAGS += -DLPL_DEF_LOCAL_WAKEUP=intervalMs
CFLAGS += -DLPL_DEF_REMOTE_WAKEUP=intervalMs
CFLAGS += -DDELAY_AFTER_RECEIVE=intervalMs
```

(b) LowPowerListening in Makefile

Listing 6.18: Using low power listening

A TinyOS application that has been compiled to TOSSIM is only a library, it requires another application to configure and run the application. The library that has been made can be used by python or c++. While c++ often is more efficient, python offers more flexibility through its interpreter. Using the python interpreter interactive mode enables the user to control the application as it is running. Python applications can also be saved in files, which can be read by the interpreter. This makes it easy to repeat simulations, and compare changes in complex simulations.

6.9.1 Debugging TOSSIM applications

Debugging applications is easy in TinyOS. The method `dbg(string outputChannel, string debugMessage)`. The second argument, `debugMessage`, is the string as it will be shown in TOSSIM. The first argument, `outputChannel`, defines which outputChannel should be used. In the TOSSIM application configuration, a user can define which output channels should be used, and where they should be sent (e.g., standard out, a file). This makes it easy to filter out messages based on which functionality you want to debug. There are also some output channels defined by the library, which can be used (e.g., AM and CRC).

6.9.2 Network topologies

TOSSIM has the ability to create network topologies. By default, the radio model used by TOSSIM is signal-strength based. In a clean application, no nodes are connected. Connections between nodes have to be manually added, this is done using the radio object from the TOSSIM library. The used method is `add(src, dest, gain)`, a connection is made from src to dest with a given gain. This

connection is asymmetric, it is therefore necessary to add connections both ways for two nodes to be able to speak together. This also enables us to simulate asymmetric connections.

TOSSIM simulates RF noise and interference from other nodes and other outside phenomenas. This is done by adding a noise trace. Some noise traces are included with TinyOS.

Chapter 7

Implementation

This section describes the use and implementation of the used TinyOS application. Section 7.1 reasons as to which RPL implementation the work should be based upon. The decision taken here meant that much work implementing features had to be done. Section 7.2 describes the required RPL features and mechanisms. An overview of the resulting RPL implementation is described in 7.3. An overview of the software process of processing DIO messages and creating upward routes are shown in 7.4, and an overview of the implemented repair mechanisms are shown in section 7.5. The implementation of a forwarding mechanism is thereafter presented in section 7.6

7.1 Choosing RPL implementation

Two pre-existing RPL implementations were available in TinyOS: TinyRPL, the RPL implementation included with TinyOS., and a limited implementation of RPL done by a previous student at the University of Oslo(UiO) Anders Taranger. Some work was done trying to make the TinyRPL work on the iris nodes, as this was not natively supported (sec: 7.1.1). Therefore, the implementation made by Anders Taranger was used as a basis, and further developed (sec:7.1.2).

7.1.1 Trying to make TinyRPL work on IRIS nodes

TinyRPL is a mature and tested RPL implementation. It interacts with the BLIP2.0 layer, which provides IPv6 support through the 6LoWPAN layer. It also provides a modularized system, making it easier to implement further functionality.

The IRIS nodes use the RF230 radio. As of the start of this master thesis, BLIP2.0 was not supported on this radio. Since TinyRPL heavily relies on BLIP2.0, it could also not be used. Investigating the TinyOS mailing-list, a git repository containing a changes made to TinyOS that were not included in the official version. Using the TinyOS from this repository, BLIP 2.0 compiled for the rf230 radio.

Being able to compile BLIP2.0 and TinyRPL, a TinyRPL test application included with TinyOS was compiled. The compilation was successful, but showed that BLIP2.0 and TinyRPL require much memory. The memory requirement for this application was 9624 bytes. This exceeded the 8kB of memory available in the

IRIS nodes. The given changes were made to TinyRPL and BLIP to try to reduce its memory consumption:

- Reduce the maximum amount of parent nodes in TinyRPL (`MAX_PARENT`) default value is 20).
- Reduce the size of the routing table used by blip (`ROUTE_TABLE_SZ`, default is 20).

By reducing `MAX_PARENT` to 1, and `ROUTE_TABLE_SZ` to 3, the memory requirement was reduced to 8456 bytes, which still exceeded the memory of the IRIS node. The memory consumption could possibly be further reduced, but not knowing if BLIP would even work even if it compiled, it was decided not to use the TinyRPL implementation. If we were to use TinyRPL, new nodes would have to be bought, or support for BLIP2.0 implemented for the rf230 radio. Due to cost and time constraints, none of these were preferable. The choice was therefore to use Anders' implementation of RPL

7.1.2 Anders' RPL

The second available RPL implementation was done in Anders Tarangers master thesis. It implemented core RPL functionality. Upward creation upward routes using DIOs, simple DIS mechanisms were implemented. No repair mechanisms were supported, and rank was not improved. This implementation chose parents by optimizing hop count. Also some code that simulated energy usage when using the TOSSIM simulator was implemented. This implementation used the AM layer for communication, and was therefore supported on the IRIS nodes.

Except for a few parts, most of this implementation was done in one component, `RplC`. The RPL implementation was mixed up with forwarding mechanisms and application-specific stuff in this one component. Many c preprocessing flags were used throughout the code to enable specific kinds of functionalities. Some of these enabled use with TOSSIM and AVRORA(another simulation software). Many of these preprocessing flags were dependent on each other, and was set manually in the Makefile. It also contained simulation specific code that simulated power usage in TOSSIM, and enabled nodes to signal that they were low on energy.

Since `RplC` was not structured, it was difficult to get an overview over its workings. This implementation was tested as was for a while, with some added functionality. The Makefile was rewritten and an accompanying configuration file created, using the configuration file removed the need to manually update all preprocessing files when making a change. Later, work started to split the different functionalities into different components, with separate responsibilities. Most of the c preprocessing flags was removed. Most functionality except for the basic sending and transmission of DIO messages were totally reimplemented. This resulted in an implementation, which were more structured, and had functionality separated into different layers.

7.2 RPL Implementation Requirements

Since a well-implemented, mature and tested RPL implementation for TinyOS already existed, the goal was not to create another one. The goal was however to create a basic implementation that provided enough functionality for the resulting scenario. Compatibility with other RPL implementations was also not needed. The RPL implementation was to be used for data collection. Some of the requirements for the RPL and the accompanying forwarding implementation were:

- Should be able to create upward routes to be able to support a data collecting application.
- No need for upward routes was needed, therefore support for DAO messages was *not* required. The only supported MOP would therefore be 0.
- The implementation should support some kind of repair. A global repair is at least required, and a local repair mechanism was strongly preferred
- The implementation should be able to detect loops. The detection of a loop should trigger a local repair.

In addition to the RPL implementation, a routing layer also had to be implemented to forward messages. Some of the requirements for the routing layer was.

- Should cooperate with the RPL component with the purpose of
  - Exchange route to reach the necessary node. In a collection application this would be root, and the necessary route would be the preferred parent
  - Allow cooperation with RPL to avoid loops by checking the RPL Packet Information.
  - Signal RPL if the parent is lost
  - Update RPL with metric information if needed.
  - Should check the availability to a parent before it is chosen.
- Some sort of reliability should be provided, merely in the form of layer 2 acknowledgments and retransmissions
- Should implement a queue for forwarding messages. Since the AM layer does not have any buffer, this is needed to make sure there is room for burst of messages.
- Should be able to forward messages over the serial connection to a connected computer.

This has resulted in a limited implementation of basic RPL functionality, and a forwarding mechanism. The RPL Implementation includes:

- Creation upwards routes using DIO messages.
- Global repair using a version number. The version number can be periodically incremented.
- Local repair in the event that a node can not select a parent.
- Using DIS messages to request DIO messages from neighbor
- Checking of RPL Packet Information to detect loops when forwarding messages.

This implementation is not supposed to be interoperable with any other RPL implementations. The focus was to make a working implementation that could be deployed and used. Some shortcuts have therefore been taken.

7.3 RPL Package

This section describes the results of the RPL implementation. An overview of the created components is shown. The interaction between them, and other used components are represented. Thereafter some core functionality is described. The network stack implementation consists of two packages: *DataSend*, and *RPL*. The RPL package is responsible for the RPL and OF mechanisms. DataSend is responsible for message forwarding. It acts like Layer 3. Both RPL and DataSend use the AM layer for communication. Since DataSend uses RPL to find the preferred routes, it is not usable by itself.

7.3.1 Overview

The RPL Package consists of some components. Some of these are specific to the core RPL functionality, and some are responsible for the OF functionality. The components implementing core RPL functionality are:

- *RplCoreP* – Is a module implementing core RPL functionality. Some of the functionality implemented in this module is:
  - Sending and receiving DIO and DIS messages.
  - Global repair
  - Local repair
  - Validate RPL Packet Information
- *RplC* – RplC is the configuration that instantiates and wires the components that are needed by RPL. RplC exports the appropriate interfaces needed for interaction with other components outside the package.
- TrickleTimerC – Implements the trickle timer. This is not a generic component, and can therefore not be used to create separate trickle timers.

The components implementing the OF functionality are:

- *VofP* –Is a module containing the OF implementation. It is an abbreviation for "Vestbø Objective Function (VOF)". This module is responsible for:
  - Store and maintain rank
  - Store and maintain the neighbor table
  - Calculate and store rank
  - Optimize routes according to the OF specification
  - Validate connectivity to parent before it is chosen.
- *VofC* – Instantiates and wires components used by VOfP. This component also exports the interfaces that VofP provides that can be used by other components.

Some interfaces are defined that enables interaction between the components used in the RPL packet, these are shown in listing 7.1 on page 70, and are:

- *Rpl* – The Rpl interface is used for interaction with components outside the package, but also for internal interaction. This interface contains commands to easily request information from RPL such as preferred parent, DAG version, rank and whether this node is a root or not. This interface is shown in listing 7.1a
- *RplInfo* – The RplInfo interface is used for communication between the forwarding layer and RPL. It contains commands to enable link information to be exchanged between RPL and the forwarding layer. This interface is

shown in listing 7.1b.

- *RplOF* – Provides an interface for interaction between the OF and the RPL Core. Among others, it provides methods to get preferred parents, provide the OF with metric information from DIO messages, trigger recalculation of parents, and reset rank. This interface is shown in listing 7.1c. This interface is not meant to be used by users outside of the package.
- *RplPacketInfo* – This interface provides the necessary methods regarding the RplPacketInfo mechanism. It validates the RPL Packet Info information in the header, and discards packets if loops are detected. This interface works between RPL and forwarding mechanisms. It is shown in 7.1d.
- *TrickleTimer* – This interface is similar to other timer interfaces in TinyOS, and will therefore not be shown. In contains commands to configure the timer, reset the interval, and start the timer. It also contains

RplC and RplCoreP

RplC is the glue between the components in the RPL package, and provides the external components to be used from other packages. An overview of RplC can be seen in figure 7.1. Three interfaces are exported, *Boot as RplReady*, *Rpl*, and *RplPacketInfo*.

The Boot interface renamed to RplReady, is used to signal users when the first preferred parent have been found. This enables users to know when communication is possible, and they can delay message transmission until this point [1].

The Rpl interface exported by RplC enables the users to request information from RPL. RplPacketInfo contains commands enabling the validation of RPL Packet Information.

RplC also uses the interface SplitControl provided by ActiveMessage. The event startDone() notifies that the radio has been started. This in turn starts Rpl: The trickle timer is configured, and started if the node is a root. DIS messages are also sent request recent DIO messages from neighbors.

RplCoreP is the core module in the Rpl package. It is responsible for sending and processing DIO and DIS messages. This is accomplished using AMReceiverC and AMSenderC. These are both instantiated twice, one for each messages type. This means that RPL messages (DIO and DIS) are sent using the AM Layer. RplCoreP uses the TrickleTimer interface to decide when to send DIO messages. Additionally, two normal TinyOS timers (TimerMilliC) are used. One of these timers is used to control DIS message transmission. The other timer is used to control the duration of the route-poisoning period when doing a local repair.

7.3.2 VofC

VofC is the configuration component for the OF functionality. An overview of VofC is shown in figure 7.2. VofP is the module implementing the OF. Three interfaces are exported from VofP, *RplInfo*, *RplOF* and *Rpl*. *RplOF* is a provided interface, which enables interaction between the OF and the RPL core. Part of this

[1] The use of the Boot interface for this purpose is kinda misleading, and should be changed. It also does not provide a way to signal users if RPL has no parent.

```
interface Rpl {
 command uint16_t getPreferredParentnodeID();
 command uint16_t getDagVersion();
 command bool isRoot();
 command uint16_t getRank();
 command bool localRepairRunning();
}
```

(a) Rpl interface

```
interface RplInfo{
 command void updateEtx(uint16_t nodeID, uint8_t etx);
 command void updateLink(uint16_t nodeID, message_t*
    msg);
 command void nodeUnreachable(uint16_t nodeID);
}
```

(b) RplInfo

```
interface RplOF{
 command uint16_t getPreferredParent();
 command bool haveParent();
 command uint16_t getRank();
 command void reset();
 command void recalculate();
 command bool updateNeighbor(message_t* msg, dio_msg_t
    * payload);
 event void parentChange(bool haveParent);
}
```

(c) RplOF

```
interface RplPacketInfo{
 command bool check(rpl_packet_info_t* packetInfo);
 command void initialize(rpl_packet_info_t* packetInfo
    );
}
```

(d) RplPacketInfo

Listing 7.1: Interfaces from the RPL package.

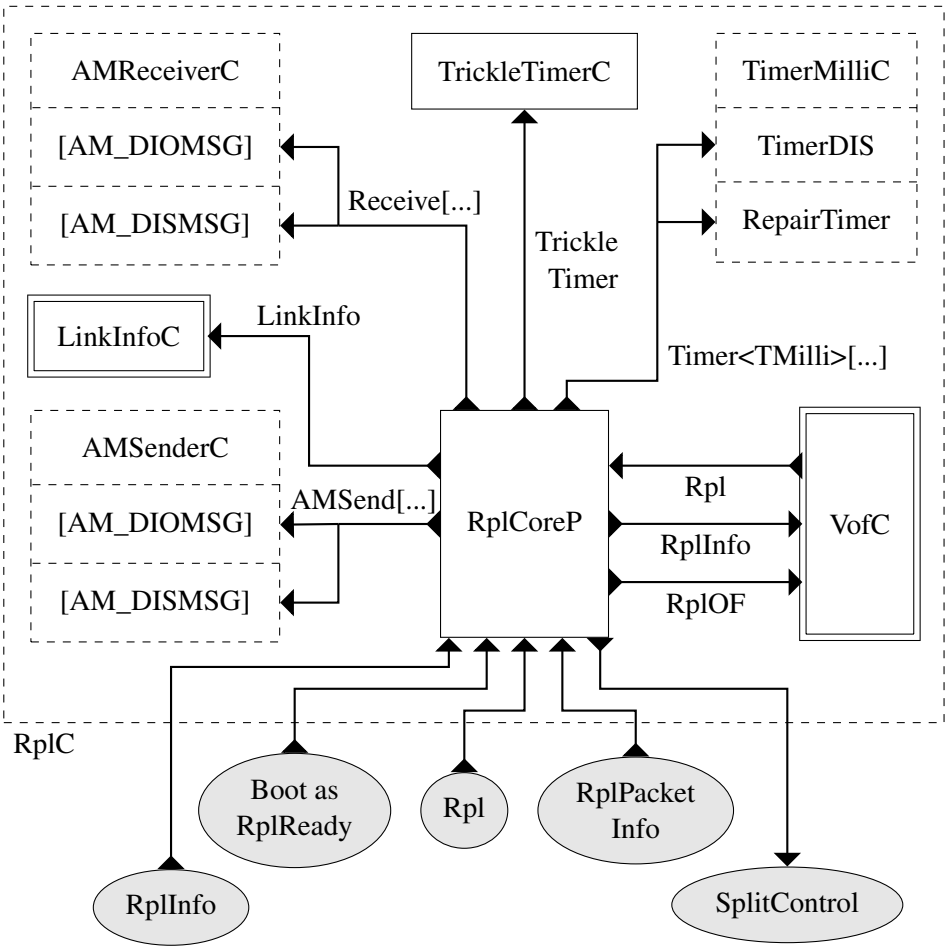


Figure 7.1: RplC configuration: Overview

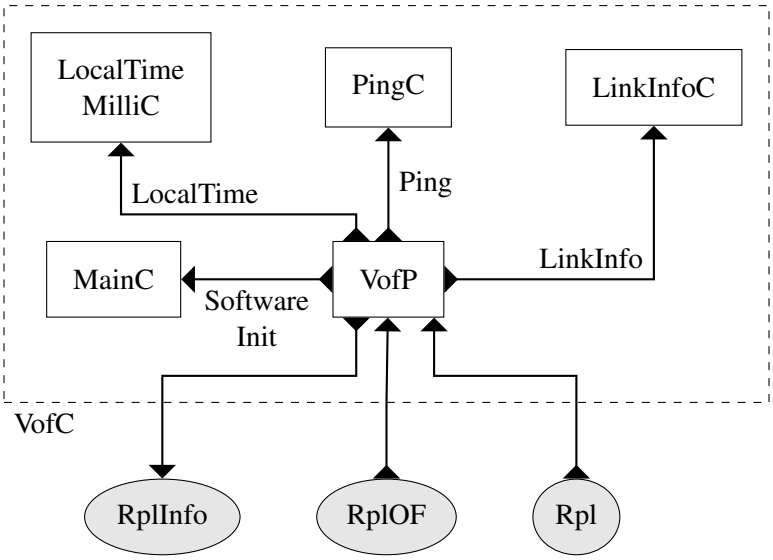


Figure 7.2: VofC configuration: Overview

work is also done by the RPL interface, which can exchange information such as the RPL version.

VofP uses SoftwareInit from MainC to initialize some data structures with during the system boot. PingC, a basic pinging component, is used to verify that the link is bidirectional before a parent is selected. This functionality also uses the LocalTimeMilliC component. LocalTimeMilliC offers a system time. This is used as a timeout for links that have failed to respond to a ping, and most likely not bidirectional, or maybe lossy.

7.4 DIO and upward routes

The RplCoreP is responsible for sending and receiving DIO messages. Together with VofP, it saves information received by the DIO messages, this information is later used to calculate the best routes. This section describes the implemented DIO message structure, and the processes used to transmit and process received DIO messages.

7.4.1 DIO Messages structure

Since this is a simplified implementation, the DIO and DIS message structure have been simplified. Some fields are discarded, while other field's size differs. The DIO and DIS message structures are defined in the file "rplcore.h" in the Rpl package.

Figure 7.3 shows the implemented DIO Messages. As can be seen, some fields have been removed, and some changed. The fields G(grounded), O, MOP(Mode of Operation), Prf(Preference), DTSN, flags, and reserved field have not been included in the implemented. The option field has also been removed. These fields were not included as the functionality offered by them were not needed. Since the AM layer is used, the DODAGID field has been reduced from 128 bit to 16 bit.

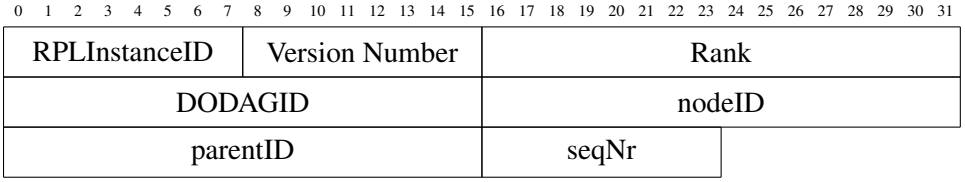


Figure 7.3: Implemented DIO structure

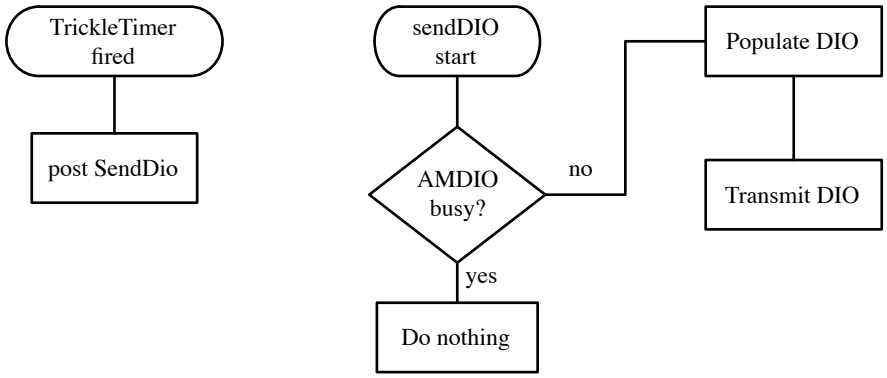


Figure 7.4: Process of transmitting DIO messages.

The version number in RPL uses a "lollipop" counter [9]. The lollipop has an initial linear phase. When this cycle is finished, the counter enters a cycle phase. This counter mechanism follows arithmetic from the RPL rfc, and other documents [44, 45]. The implementation version number counter is only linear. The test network was deployed for a longer time than anticipated. Because of this, and the fact that the version number had to be updated more regularly than thought[2], the version number counter was increased to 16 bit for a while. Some addition fields were added, *nodeID*, *parentID*, and *seqNr*. These were leftover fields from the old implementation that never got removed. As this information is either accessible from the header of the packet, or not necessary, these fields should be removed.

7.4.2 Transmitting DIO messages

The process of transmitting DIO messages is very simple. It is triggered by the trickle timer. In the case that the node is a root node, the trickle timer is started at boot. If this is not the case, the trickle timer is started when the node gets a preferred parent. The process of DIO transmission is shown in figure 7.4.

7.4.3 Receiving and processing DIO messages

The process of receiving and processing messages is more complicated than the transmission of DIO messages. As DIO messages are received, they have to be processed. The processing of a DIO message starts when the receive method provided by the AM layer is signaled. An overview over what happens is shown in

[2]There was a bug where the timer of the first deployed nodes acted up. Since some of the nodes were physically unavailable, it was impossible to reprogram them. They did however trigger a data send as the version number increased.

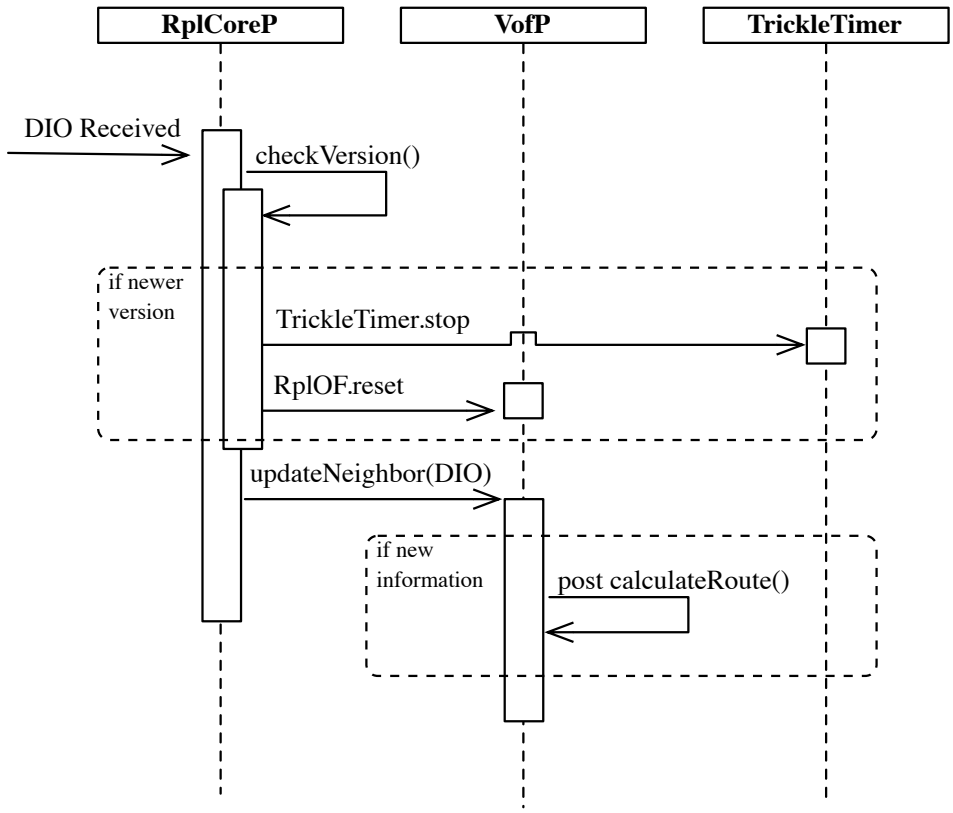


Figure 7.5: DIO Receive Sequence for a non-root node.

7.5 . The first step is to compare the version number of the received DIO with the current version. If the DIO version is newer, a global repair is triggered:

- For non root nodes, the trickle timer is stopped and the process of global repair is started. This process is described in section 7.5.1
- If the node is a root node, the version number is increased to be larger than the received version number. The presence of a version number in the network larger than that of the root, indicates an error of some sort. This therefore makes sure the whole network is updated.

For root nodes, the processing of DIO messages is finished after the comparison of version numbers have been done. The following process is only done for non-root nodes.

The method updateNeighbor() in VofP is responsible for further processing of DIO messages. It compares the contents of all DIO messages against the neighbor table. It updates the neighbor table according to the new DIO message. If a node is new, the properties received from the DIO message is used. If the node already exists, the information is updated. For existing nodes, the information is updated. This involves calculating the average for link information. If certain changes have been done to the neighbor table, or a new node have been inserted, the calculateRoute() task is posted.

CalculateRoute() is the initial method in the parent selection process. This process is defined in VofP, it is shown in figure 7.6.Initially, the version number

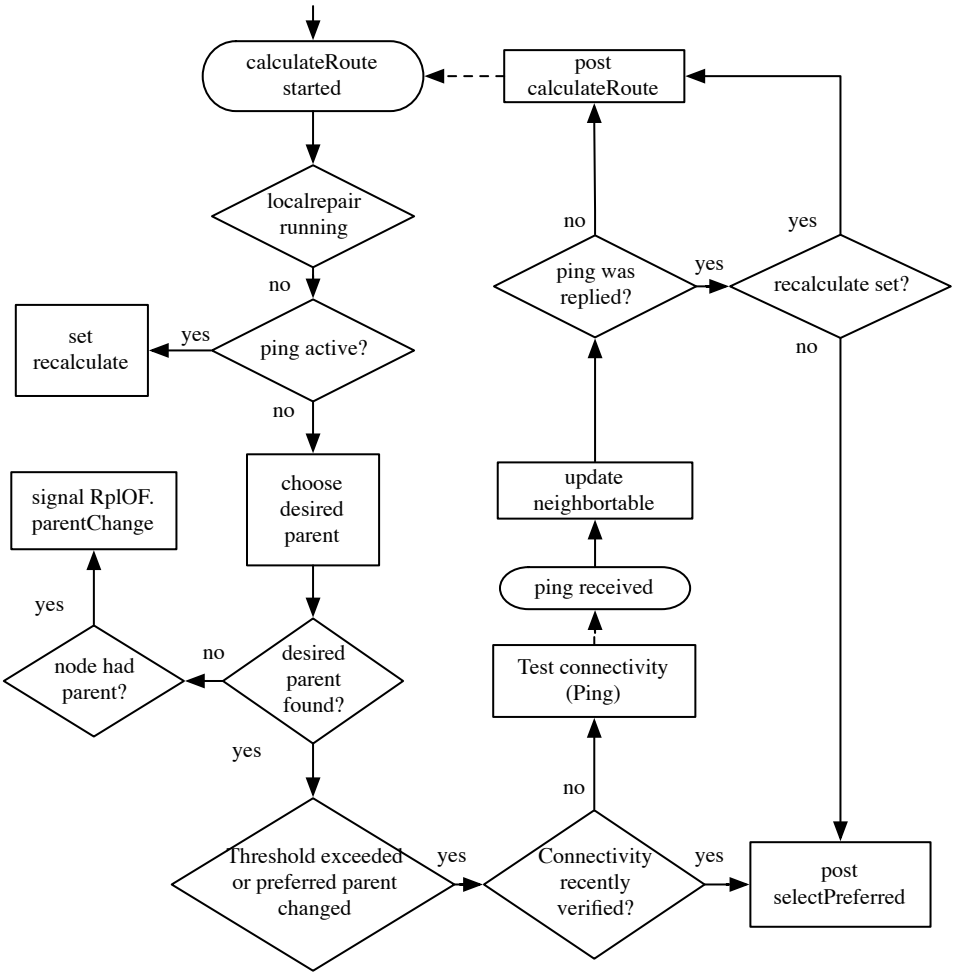


Figure 7.6: Process of selecting a parent (calculateRoute)

in the received DIO is compared to the nodes current version. The steps involved in this task afterward are:

1. A check is done to see if a local repair or ping is in process. If this is the case, the task exits immediately. The task will be reposted later when either of these processes finishes. If it exists due to an active ping, a flag is set which instructs the ping process to repost calculateRoute laterThis ensures new information is considered before a parent is selected. Since Ping is a split-phase process, a calculateRoute task can be started while waiting for a Ping reply. We do not want this calculateroute to be run, as it can interrupt the ping process.
2. The process of finding the best candidate from the neighbor table is started. The currently best candidate during this process is called the "desired parent". The process of finding the desired parent starts by iterating over the neighbor table. Each node is checked against a list of requirements that decides if it is a usable parent or not. These requirements are:
   - The nodes rank must not be INFINITE_RANK
   - The resulting rank must be less than the current rank

- The link cost must not exceed MAX_LINK_METRIC. This discards links that are of poor quality
- The node must have the same version number. A nodes version number is always the most recent version number found in the network
- Nodes that have been recently pinged and found to be unreachable are not to be considered. This interval is set by PING_CHECK_-THRESHOLD.

The outcome of this can be:

- No desired parent can be found. In this situation we do not have any usable parents. The RplOF.parentChange(FALSE) is sent if we previously had a preferred parent. This informs RPL that no parent is found. RPL can then start the appropriate methods for this situation. At this moment, this triggers a local repair. This also exits the calculateRoute task.
- During the selection of the desired parents, a node is selected as the desired index if it provides a better resulting rank than the currently selected preferred parent. If the resulting ranks are equal, the node with the best signal strength is selected as the desired parent

3. If a desired parent have been found, it is compared to the current preferred parent. If the rank increase does not exceed the threshold (PARENT_-SWITCH_THRESHOLD) from the OF configuration, the old preferred parent is chosen. This is part of the hysteresis function from MRHOF. There are some exceptions to this rule:
   - The preferred parent rank is larger than the current rank. This means that the preferred parent is invalid and cannot be used.
   - The preferred parent is the desired parent and its rank have been decreased. In this case the process continues to decrease the advertised rank.
4. If connectivity to the desired node has not been validated recently, a Ping process is started. This is not done if the desired index is the same as the parent node. The ping process has two options: a timeout, and the amount of retries. The neighbor table is updated with the ping information. The ping process has two outcomes:
   - A ping reply was not received. The link is therefore not verified to be bidirectional, and should not be used. The desired parent node is invalid. This restarts the calculateRoute process.
   - A ping reply was received. This verifies that the link is bidirectional. If someone have reposted the calculateRoute, new information is available, and the process of finding a parent restarts. If this is not the case, the desired node is chosen to be the preferred parent. This is done by selectPreferredParent()

   If a link has been previously verified within a given period (PING_CHECK_-THRESHOLD), the Ping process is skipped, and selectPreferredParent is posted right away.
5. selectPreferredParent() is posted. It updates the current preferred parent, updates the rank, and sends the appropriate signals to inform RPL of changes.

7.5 Repair mechanisms

RPL specifies two types of repairs: *global* and *local* repair.

The global repair updates the whole network. All the nodes are free to choose the rank they want. This implementation supports global repair.

RPL allows a node to perform local repair by increasing its rank by an amount of MaxRankIncrease compared to the least advertised rank in the current version. MaxRankIncrease is decided by the root node. This variable ensures that node are not too greedy. This type of repair has not been implemented, but work on this was started. The second local repair mechanism allows a node to set `INFINITE_RANK`. After this it has to poison the network to make sure it is not selected by its sub-DODAG. After this it can freely choose the rank it wants.

7.5.1 Global Repair

Root node

Only the root can trigger a global repair. The root goes through the following process when initiating the global repair:

1. The version number is incremented. If a version number has been received in a DIO from a neighboring node, the version number is incremented to be larger than this.
2. The trickle timer is restarted to quickly spread the new version to the neighbor nodes.

As can be seen, this process is very simple for the root node.

Non-root nodes

For non-root nodes the process of a global repair starts when receiving a DIO message containing a newer version number than the nodes current version number. In the implementation, a node joins a new version as soon as it has been detected. An overview of the global repair in non-root nodes can be seen in figure 7.7, it is:

1. A DIO with a new version has been processed.
2. The Trickle timer is stopped.
3. The current version number is updated.
4. The call RplOF.reset() is called. This causes to RplOF to reset its rank and haveparent property. This enables the OF to choose freely without any restrictions on rank.
5. The OF sees the new version number and thus chooses a new parent.
6. When the OF signals RPL that a parent have been found, the trickle timer is reset and started.
7. The DIO messages that are sent now use the new version number. This causes members in the sub-DODAG to update to the new version

7.5.2 Local repair

Local repair is different from global repair in that it is only specific to a certain node at any given time. This does however not mean that other nodes are unaffected by local repairs. Nodes in the local area and sub-DODAGs are affected, but other parts

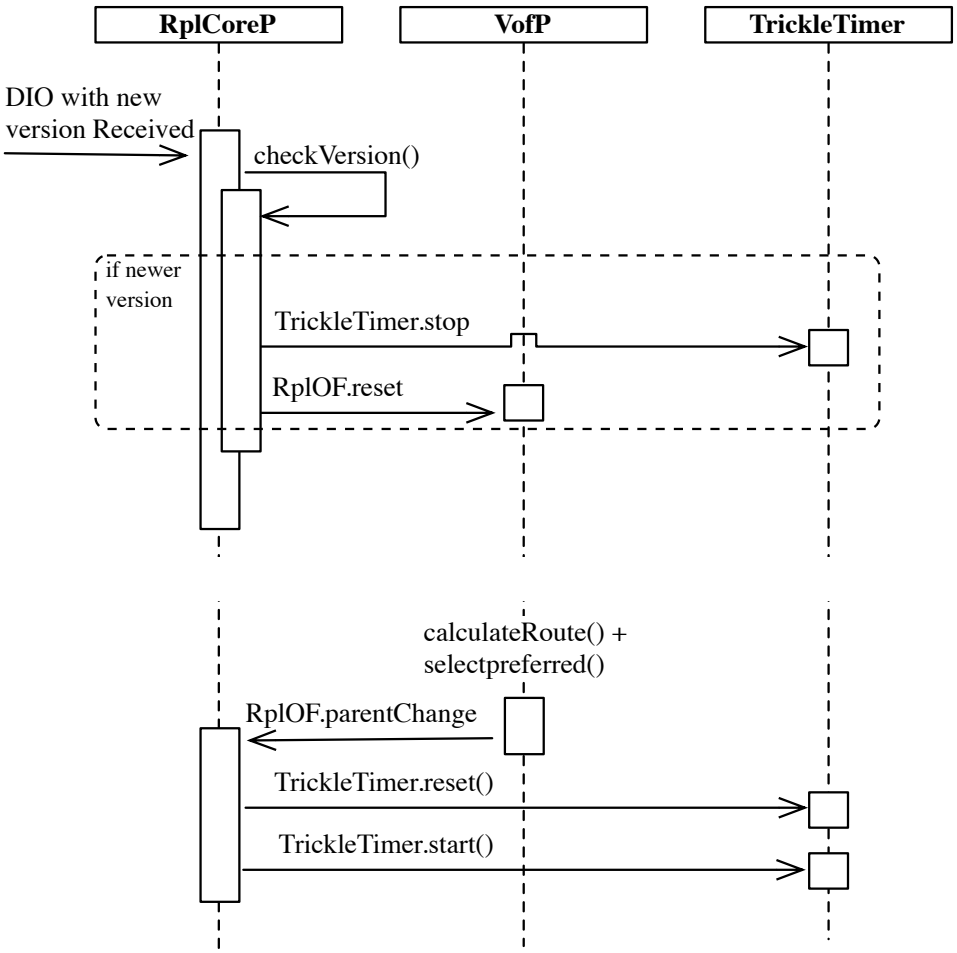


Figure 7.7: Global repair in a non-root node

of the network are not. Local repair is initiated by the node in need of a repair. This is done in RplCoreP, and is triggered by a signal from RplOF that a parent has not been found. This signal is only triggered if preferred parent already existed. The interaction between components during the local repair is shown in figure 7.8. The flow of the mechanism is:

1. A signal is sent from RplOF that no new parent is found. This signal is only sent if a node had a parent, but now can not find any new.
2. The TrickleTimer is stopped.
3. The DIS timer is started. When this timer expires, the node sends DIS messages. This in turn leads to new DIOs, which update the neighbor information. This delay should be very short if used.
4. The repair timer is started with a length of LOCAL_REPAIR_DELAY. This timer enables the node to find a new parent within its current rank restrictions from the new information from the received DIOs.
    - If a node is found during the the repair delay. The delay timer is stopped. This also stops the local repair process.
5. If a new parent is found before the poison delay timer has finished, the timer is stopped and normal operation continues. If no parents are found, the timer triggers, and route poisoning is started.:
    (a) The RplOF.reset command is called. This sets the rank to INFINITE_-RANK.
    (b) The trickle timer is reset and started.
    (c) A Timer is started to control the duration of the route poisoning. When this timer is running, RplOF is not allowed to choose any parents.
    (d) DIO Messages with INFINITE_RANK is transmitted. This poisons this nodes entry in receiving nodes neighbor tables other. Nodes in the sub-DODAG thus have to find a new parent.
6. When the poison timer is finished, a route recalculation is triggered. The node is now at INFINITE_RANK. This enables the node to consider all possible parents.

7.6 Message forwarding

A simple layer 3 forwarding/routing layer has been implemented. Active-Message (AM) is the standard TinyOS layer 2 framework. It offers single hop communication. An independent layer 3 forwarding mechanism was not found in TinyOS. An exception to this is BLIP2.0, which we were unable to use because of compatibility issues. Layer 3 communication was thus implemented in the *DataSend* package. DataSend provides a very simple forwarding mechanism. As its only purpose was to enable data-collection, a route to the root node was the only routing information needed. A separate routing table has therefore not been created. DataSend uses the preferred parent for all forwarding. The addresses used in Layer 3 are currently identical to the AM addresses. Additionally, an address have been assigned to the computer connected to the root node through the serial port.

DataSend does not currently use any form of multiplexing or message types. It can therefore currently only be used by one upper layer component.

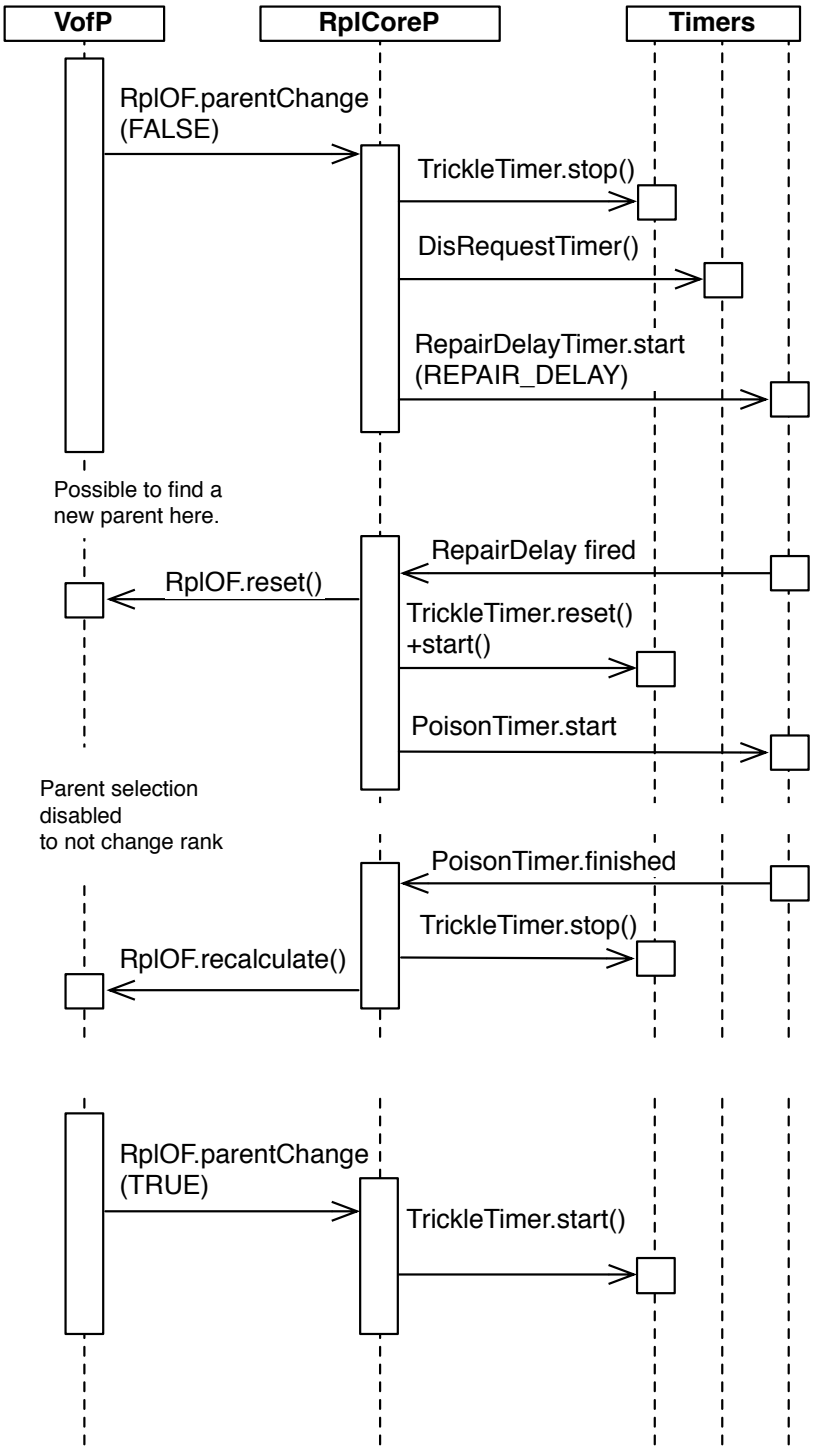


Figure 7.8: Local repair sequence diagram.

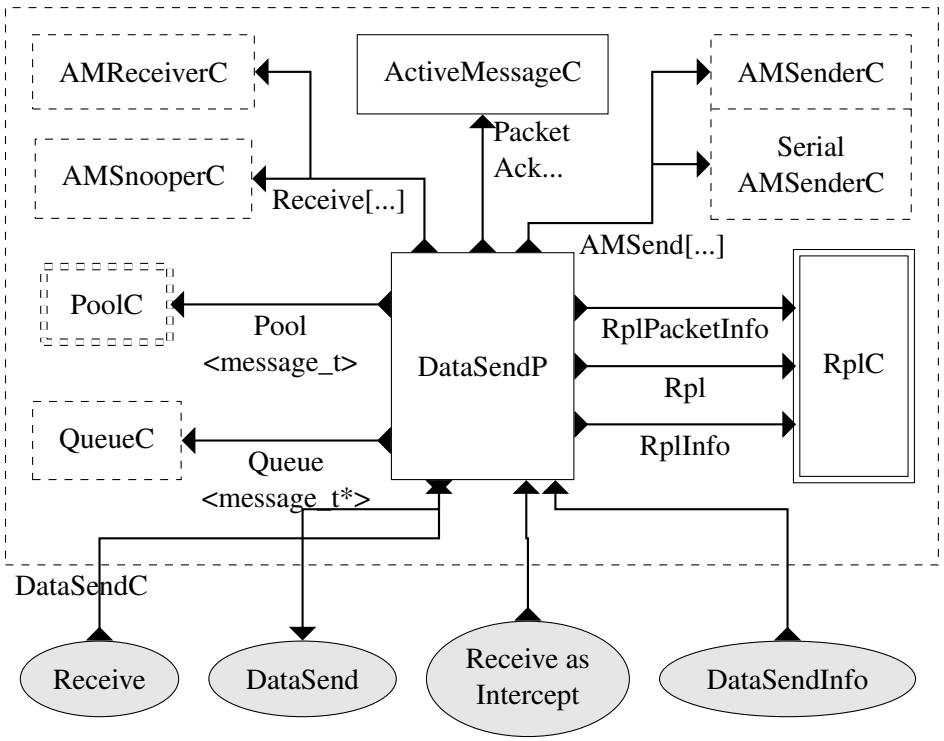


Figure 7.9: Overview over DataSendC

7.6.1 Overview

The DataSend package is a relatively small package. It consists of two components, *DataSendC* and *DataSendP*. DataSendP is a module containing the logic in the application, while DataSendC wires the necessary components together, and exports usable interfaces for users outside the package. An overview of DataSendC is given in figure 7.9. DataSendC exports three interfaces, *Receive*, *DataSend*, and *Intercept*. DataSendP uses the interface Rpl from RplC to get preferred parent and routing information. The RplPacket from RplC is used to approve packets that are to be forwarded according to the RPL packet information. RplInfo from RplC is used to update link information in Rpl. This includes information about received packages, and experiences made when forwarding. This enables path costs like ETX to be updated.

DataSendP uses some components for transmitting and receiving messages. AMSenderC is used for radio transmissions, and SerialAMSenderC is used for serial transmissions. AMReceiverC is used to receive messages from the radio destined for this node. Additionally, AMSnooperC is used to receive messages intended for other nodes. This enables more link information updates to RPL.

DataSendP uses QueueC as a messages queue. QueueC offer simple FIFO queues. The queue contains pointers to message_t buffers. Since dynamic memory allocation is strongly discouraged in TinyOS, a pool (PoolC) is used for message_t buffers. PoolC statically allocates a given amount of message buffer objects, which can be retrieved or inserted into the pool. Instead of dynamically allocating a message buffer, a message buffer is retrieved from PoolC. PoolC holds a given amount of message_t objects, this amount is given at compile time.

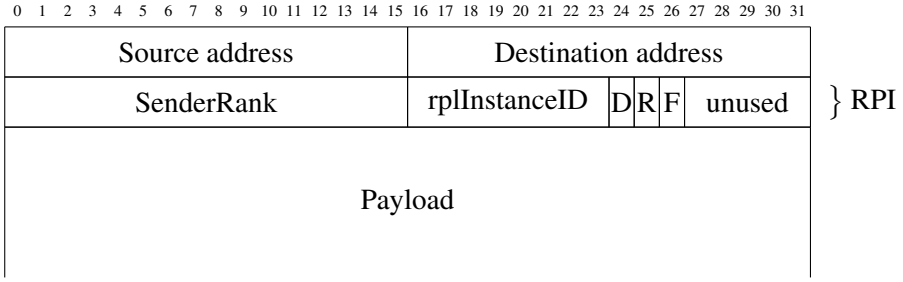


Figure 7.10: DataSend message structure

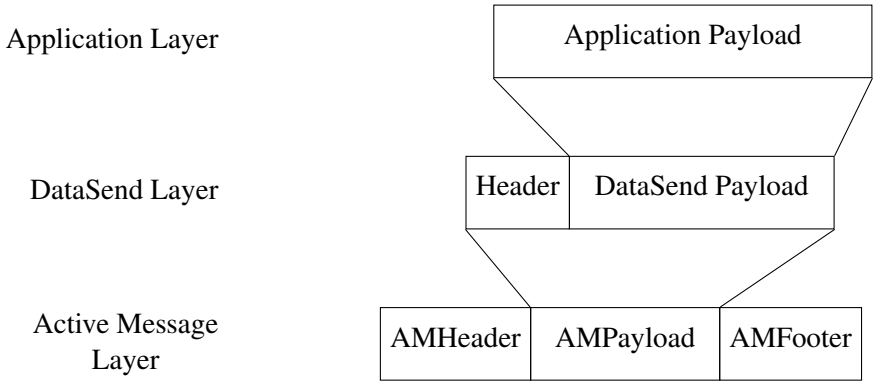


Figure 7.11: Packet layering

Message format

DataSend inserts its own header into a message_t payload area, the message structure used by DataSend is shown in 7.10. It consists of a layer 3 source and destination address, the RPL packet information, and application payload. Figure 7.11 shows how the layer encapsulates the packets.

7.6.2 Sending messages with DataSend

Messages are sent with DataSend using the DataSend interface. This interface is shown in listing 7.2a. This interface is very similar to the AMSend interface used with the AM layer, but has a few changes. First, there is no sendDone command. The upper layer has therefore no way of knowing if, or when, a package has been successfully sent. For this reason, a local message_t* buffer is not kept like with the normal AMReceive. The getMessage command is introduced for this reason. This command returns a message buffer from the DataSend pool. Care must be taken in the application to not drain the pool for messages, as this will stop DataSend from also being able to forward messages.

1. DataSend.send(...) command is called from the application layer. This command specifies destination address, message buffer and payload size.
2. DataSend checks if a preferred parent exists, that the payload size does not exceed the maximum allowed payload size, and that the message queue is not full. If either of these fails, an error is returned from the send message, the message is put into the pool, and the process stops.

```
interface DataSend {
 command message_t* getMessage(uint8_t len);
 command void* getPayload(message_t* msg, uint8_t len)
     ;
 command uint16_t getDestination(message_t* msg);
 command void setDestination(message_t* msg, uint16_t
     addr);
 command uint8_t maxPayloadLength();
 command error_t send(uint16_t dst, message_t* msg,
     uint8_t len);
}
```

(a) DataSend interface

3. DataSend initializes the message with source and destination address.
4. The RPL Packet Information is initialized by calling RplPacketInfo.initialize().
5. The message is put enqueued for transmission (MsgQueue). This queue is used for all messages awaiting transmission. If this is successfully done, the task queueSend is posted, and SUCCESS is returned. If not, the message is put back into the pool, an error message is returned and the process is canceled.

The flow of queueSend is:

1. QueuePost exits if the outgoing queue is empty
2. If a transmission is active, the task is reposted, and exited.
3. The destination address of the message is checked.
   - If the destination is the computer and the node is root, the message is transmitted on the serial port with an acknowledgment requested
   - If this is not true, the message is forwarded to the preferred parent on the radio link.

   Since radio and serial both use AM, this is done by calling send on either the serial component, or the radio component.

Since send is a split-phase, the succession of either of the sendMessages is signaled with a sendDone event. Both of these sendDone follow the same flow:

1. If an acknowledgment was received, the message is ok.
   (a) The etx average variable is updated according to according according to the number of retransmissions
   (b) The message object is removed from the top of the queue.
2. If the message was not acknowledged it is retransmitted. The max amount of retransmissions is given by NOACK_RETRY.
3. If the maximum amount of retransmissions have been done, the RplInfo.nodeUnreachable() command is called. This notifies RPL that the node is not accessible, and the link quality information in the Rpl neighbor table is reduced. This also triggers a route calculation.
4. When the message has been successfully transmitted or met the maximum amount of retries, it is removed from the queue and put back into the pool.

7.6.3 Receiving and intercepting messages

Two receive interfaces are exported from DataSendC: *Receive*, and *Intercept*. The Receive interface is implemented in TinyOS and is shown in listing 6.13 on page 51. The receive interface is used for messages which are destined for this node. The intercept interface is used to intercept messages that are to be forwarded by the node. This enables the application to alter messages that are forwarded. This was implemented to be able to record link information as a message traversed nodes towards the root node. The buffer-swap policy is still used with Receive, but is slightly changed for Intercept. In Intercept, buffer swap is not used. DataSendP ignores the return value of this signal, and the same message buffer is used. The implementer of the intercept interface must therefore not do any processing whatsoever on the message after the Intercept.receive() method has finished.

An overview of what happens on message reception is:

1. DataSendP receives a receive() signal from the AM layer.
2. DataSendP signals RPL that a packet has been received. This is used to update link information.
3. If the message pool is empty, or the receiving queue full, the message is discarded, and the message buffer sent back to the AM layer.
4. If the message queue is empty the task exits immediately
5. The messages destination is checked.
6. If we are the destination, the receive method in the receive interface is signaled. The returned message buffer is put back into the pool.
7. The Rpl Packet information is checked. If this check returns an error, the message is discarded and the buffer put in the pool, and the process ends
8. If we are not the destination, the receive method in the Intercept interface is signaled, this allows the application layer to intercept messages that are forwarded.
9. If the message queue is full the message is discarded and put back into the pool. The method then exits.
10. The message is enqueued to the message queue, and the task queueSend() is posted.

7.6.4 Snooping around for RPL

DataSendP also uses the Receive interface from AMSnooperC. AMSnooperC is used to snoop messages that are not received by AMReceiverC. The message received from this interface is used to update link information in the neighbor table on received data packets.

7.7 Hardware dependencies

During the development of the application, TOSSIM was used heavily to test that the code was working as wanted. Some HAL interfaces have been used in order to be able to acquire certain information. To be able to compile the application seamlessly for the platforms used (TOSSIM and IRIS), some wrappers were created that made this work easier. The two most important wrappers are LinkInfo and Sht11Wrapper.

LinkInfo

Link information has been used in Rpl for finding link quality, and also measure by the application. This information is stored in the metadata field in the message_t buffers. Specific interfaces have to be used in order to access these fields. These interfaces differs between radios, and are not exported by the hardware independent ActiveMessageC.

The LinkInfo interface provides a common interface to access the radio link metadata. Used with the LinkInfoP and LinkInfoC component, it enables access to link information for iris and tossim platforms.

Sht11Wrapper

Is used as a wrapper for the sensors that were used. The sensors are used on the IRIS platform, while random values are used for the tossim platform. The configuration Sht11C wires these depending on the platform used.

RplPowerC

RplPowerC is a wrapper used to get voltage. This returns the voltage for the IRIS node, and a random value for TOSSIM, as no concept of energy exists here.

7.8 Application using RPL and DataSend

In order to use the Rpl and message forwarding code in DataSend, an application was built on top of this. This application was made to collect information from a sensornetwork in a church. This scenario will be described more closely in chapter 8. This application would use the MTS420 sensor board, and report data and humidity information to the base station node with 5-minute intervals. Additionally, this application would also gather link information for the first hop taken by the packet, and an average amount of transmissions before successfull acknowledgment. An overview over this application, called MeasureAppC can be seen in figure 7.12.

The MeasureAppC receives the booted signal from RplReady, meaning that RPL has found a preferred parent. This triggers the timer in MeasureAppC to start, and the sensors to be read. Each sensors read method in parallell as described in section 6.7. A timer is started that waits for a little time, enough for the sensors to be read. When this timer fires, a packet is sent using DataSend.send(). In addition to humidity, temperature and voltage readings, this message includes the preferred parent node id, the DODAG version, and the ETX experienced by DataSend. A new timer is now started with the data-reporting interval, which triggers a new sensor reading in a given amount of time. The application payload can be seen in figure 7.13.

MeasureAppC utilizes DataSend.intercept(...) in order to also record information about the next hop. Upon forwarding a message, DataSend.intercept(...) is signalled. MeasureAppC logs the RSSI and LQI values from the packet if it is the first hop for the message.

The resulting stack can be seen in figure 7.14

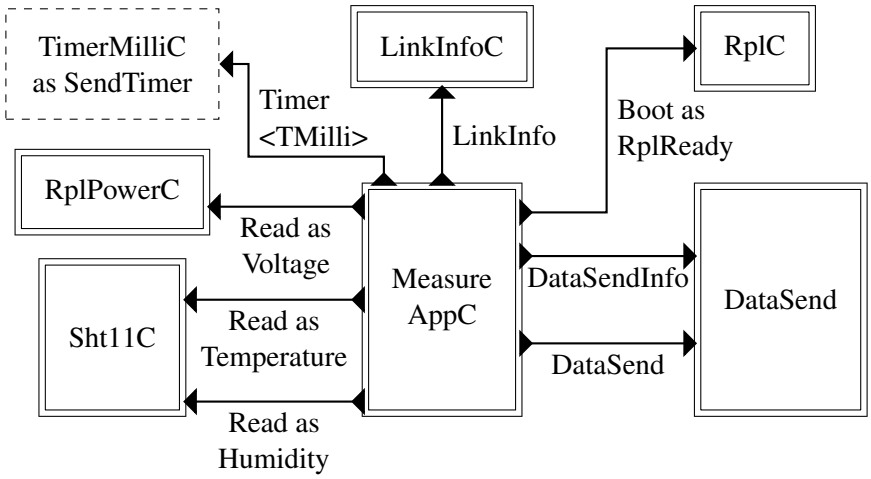


Figure 7.12: Overview over MeasureAppC

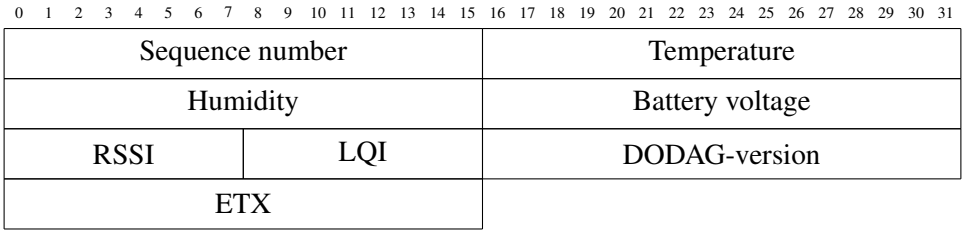


Figure 7.13: MeasureAppC application payload

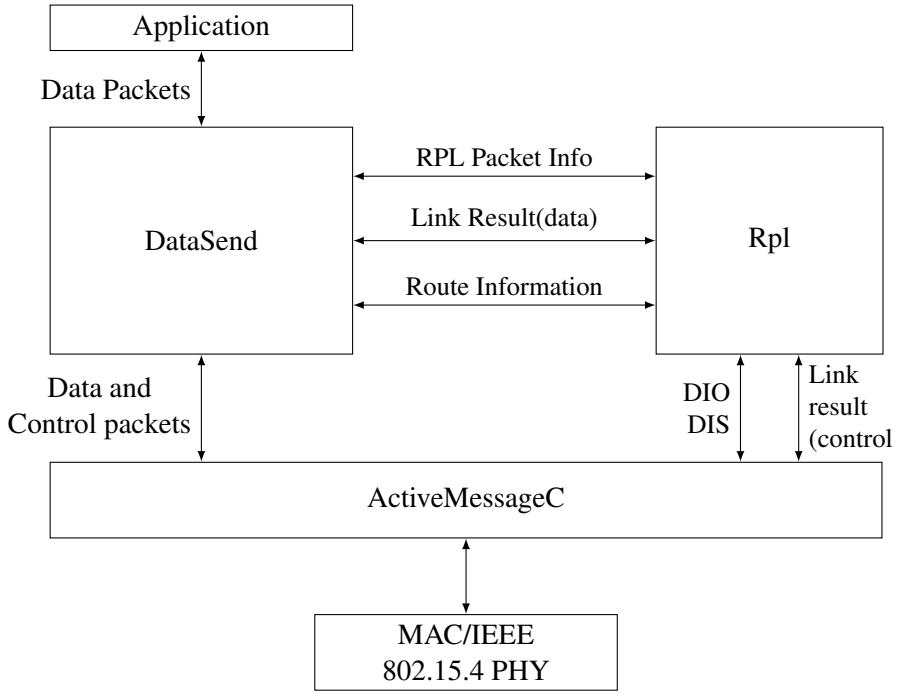


Figure 7.14: DataSend Stack.

Chapter 8

Deployment, results, and analysis

This chapter will introduce the environment and the use scenario for the wireless sensor network. Results will also be shown and analysed.

Section 8.1 describes the church in Laksevåg, and why a WSN is needed in this church. The deployment of nodes in the church is described in section 8.2. Some practical challenges that were met are described in section 8.4. Results and analysis of the running network is thereafter described in 8.5

8.1 Church of Laksevåg

The church of Laksevåg is a wooden church originally built in 1874. Some renovations and extensions have been made to the church later (1919-1935). A picture of the church can be seen in figure 8.1. The church consists of a main part, where the church spire is. This is the part of the church where the deployment of the WSN will take place. An extension to the main part can be seen at the left side at the main church. The church is quite old, and amongst its inventory are old and valuable paintings, and a church organ, all of which contain historical and cultural value, and should be taken good care of.

Environmental challenges

Like many old buildings, the church in Laksevåg is not very well isolated. Electrical heaters are used in order to keep the rooms warm; this in turn causes the air to become dry. As the electrical heaters are used much during cold winter periods, this actually makes the air so dry that the paintings are in danger of drying up, and the organ can take damage. To prevent this from happening, an air humidifier is being used to increase air humidity. This in turn creates a different problem. When warm and humid air meets cold surfaces, condensation is created. This happens when the warm and humid air ascends and meets the cold roof. This condensation might cause the building material to rot. Care must therefore be taken that the air humidity is kept within acceptable values: humid enough to protect paintings and the organ, but not too humid as this can damage building structures.

A good monitoring solution would be able to provide valuable information in order to optimize the use of the air humidifier. Temperature and humidity information from several critical locations could be monitored and provide

Figure 8.1: Church of Laksevåg
Photo: wikipedia (HMPinnsvinet)

information needed for both of these interests. A WSN can easily provide the needed information. Nodes can be placed in the critical areas, and provide information about temperature and humidity. Due to the small size of nodes, and no need for infrastructure, a test could easily be deployed. A scenario like this was perfect for a test, it provided valuable information for the church environment, and was an excellent chance to get an idea how the implementation worked.

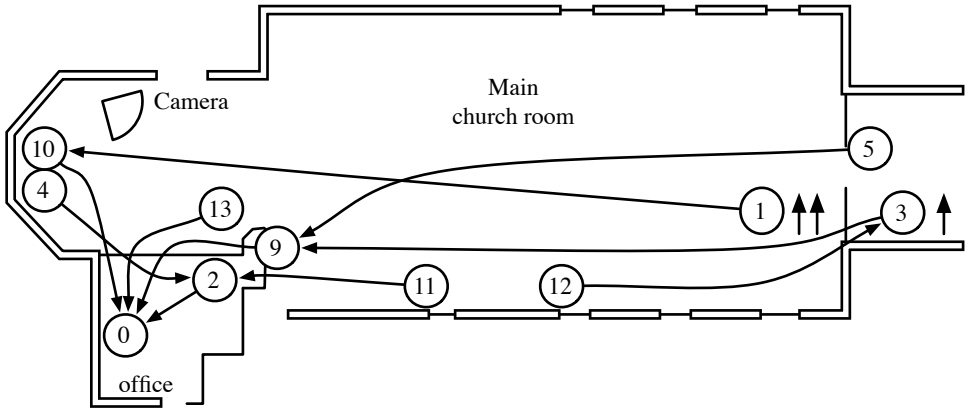
8.2 Node deployment

In order to gather the needed data, a number of nodes were placed throughout the church. Two factors decided the location of these nodes. Some nodes were first placed in locations were monitor information was wanted. Some of these locations are (see figure 8.2):

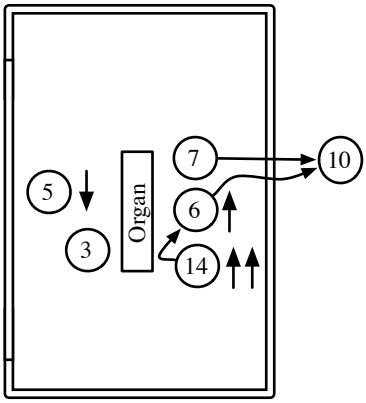
- Ceiling – A sensor node was placed above the ceiling of the church room, but beneath the upper roof. This is a location that could be badly affected by humidity (node 1).
- Art and organ– Some sensor nodes were placed close to locations where the air should not be too dry (node 3, 7, 10).
- The church bells - One sensor node was placed all the way up by the church bells (node 14).

The other critical location was the location of the base station. Electrical power, and more size was needed for the base station, as it would be connected to a computer. An Internet connection was also preferred to be able to read monitor data in real time from other locations. The computer was therefore placed in an office next to the church main room. The base station (node 0) was located right above the computer.

In order to ensure good connectivity, nodes were spread out between the already placed nodes, and the base station. This would increase the chance of good connectivity, as physical distance between nodes was reduced. A total amount of



(a) 1st floor



(b) 2nd floor

(c) Main Church room

Figure 8.2: Node locations

Node locations are shown here. Arrows at the side of a node indicates height difference from the shown level. The amount of arrows indicates a rough description as to how many levels above or below a node is. An arrow pointing up means the node is above, while an arrow down means below. In (c), the church organ on the second floor can be seen in the back of the room, above the exit. The most used hops from implementation 3 are marked with arrows between nodes. Note that this is not necessarily a topology that has actually been used.

13 nodes were deployed, excluding the base station. All nodes except the base station were equipped with mts420 sensor boards.

The location of the nodes can be seen in figure 8.2. A node is marked as a circle, with the node id denoted inside. The arrows show the preferred parent that has been used to forward most packets for each node. Node 0 is the base station, and is connected to a computer with an Internet connection. Node 14 is in the spire with the church bells, and is the node furthest away from the base station. The spire can be seen in figure 8.1.

8.3 Base station operation

8.3.1 Storing data from the nodes

A computer application was developed to receive and store the received data from the WSN. The application was written in python. It uses message interface generator to access the necessary fields in packets that are received using serial forwarder. Serial forwarder also allows us to check debugging messages from the base station simultaneously.

The data gathering receives and stores information using the producer-consumer principle. Two threads, and a shared queue is used. The producer thread receives packets and enqueues them into the shared queue. This thread signals the consumer thread when a new packet has arrived. The consumer thread reads the queue, and stores the information in three different ways:

- Comma-separated value (CSV) files are created for all received packets.
- A database is updated through the internet connection.
- CSV files are created for information that did not make it to the database. This allowed for easily updating the database if connectivity was disrupted, or other failures occurred.

The information stored in the database consists of raw values directly from the sensors. This removed the possibility of wrongly calculating the human read values. In order to reduce processing time needed to convert these values every time information was requested from the database, a materialized view (MV) were created. Insert to the main database triggers updates to the MV. The MV is thus an always-updated copy of the main database, but with pre-calculated values. This greatly reduces loading time.

8.3.2 Web interface

A web interface was created in order to provide easy access to temperature and humidity. This interface is shown in figure 8.3. The interface shows a temperature graph, and a humidity graph, containing information from all nodes. These graphs are made using the highcharts javascript Application Programmin Interface (API). The user is able to zoom the graphs, and change the timescales. The information shown in the graph is averaged to reduce the amount of data it takes to load the site. The graphs shown are updated as the user changes timescale or zooms. The graph thus provides reasonable time resolution even if the user zooms in, but does not need to load this information if a big timescale is used. Users also have the ability to turn on/off the display for any given node.

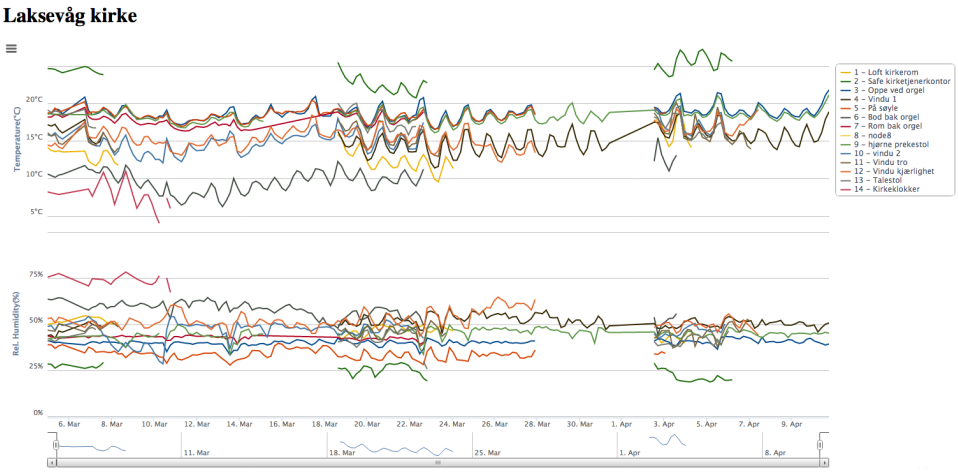


Figure 8.3: Screenshot of web interface presenting measured humidity and temperature.

8.4 Practical challenges

8.4.1 Challenges when deploying the network

During the deployment of the network some practical issues were encountered. These were annoying, but mostly fixable.

Timer gone wrong

As the nodes were deployed, every node was turned on when being placed in its locationA web interface was created in order to provide easy access to temperature and humidity. This interface is shown in figure 8.3. The interface shows a temperature graph, and a humidity graph, containing information from all nodes. These graphs are made using the highcharts javascript api. The user is able to zoom the graphs, and change the timescales. The information shown in the graph is averaged to reduce the amount of data it takes to load the site. The graphs shown are updated as the user changes timescale or zooms. The graph thus provides reasonable time resolution even if the user zooms in, but does not need need to load this information if a big timescale is used. The user also have the ability to turn on/off the display for any given node. The base station was the last node that was set up. This caused the nodes to promptly wait for DIO messages and establish routes before initializing any data transmissions. As the base station was turned on, information was received from all nodes, but only once. A different timer interval had been used during testing of the nodes. As a last step of making the nodes ready, the timer value had therefore been changed. The timer was explicitly set as an unsigned by appending the letter u after the number. This did not work very well, and the timer did not seem to be triggered at all.

A new timer value was set which made the nodes operate as intended. The nodes thus had to be reprogrammed. This was possible for all nodes, except the node in the church bell tower. Only some people were allowed to access this area for security reasons. Therefore, the node in the tower was not reprogrammed.

Since the data reporting also were triggered by RPL global repair, the global repair timer was reduced in order to be able to get some information from this node.

Computer crashes

A raspberry pi was first used as the computer that was connected to the base station. After a couple of days, it decided to crash. Similar crashes had already happened a couple of times prior to deployment. Another computer was therefore brought to the church and replaced the raspberry pi.

8.5 Experiences with the routing protocol

Two versions of the RPL implementation were used during the deployment of the network. The first version ran for about 2 weeks. Experiences from this test led to some improvements that were deployed during a second run. Some bugs were found after the second implementation was deployed. These have later been improved and been tested in TOSSIM, but are not yet tested in a real deployment. Additionally the calculation from LQI to ETX was changed, as this had been done wrong in the first test.

8.5.1 The first implementation

The first implementation that was deployed did not have all the functionality that has been described earlier. Some details of the first deployment is listed below:

- DIO messaged were used to update the neighbor table and trigger the recalculation of the preferred parent.
- Neighbor information was updated from received DIO messages. Updating this information also triggered recalculation of the preferred parent.
- Global repair was implemented and working as intended. Global repairs were triggered regularly be the root
- Local repair was not enabled because of some uncertainty as to how this was working.

In this deployment, the minimum trickle timer interval was 256ms, and the maximum 1.15 hours. LPL was enabled for normal nodes with a sleep period of 64ms. Parent For both deployments the root node had a sleep interval of 0, but used LPL techniques when transmitting messages to make sure they were received by LPL enabled nodes. Both implementations used a data-reporting interval of 5 minutes. Implementation errors caused the calculated ETX to be 1 when LQI was larger than 240, and 2 when the LQI was lower than 240. Both implementations used a parent switch threshold of 2.

Issues in the first implementation

Changes in links are one of the typical characteristics of LLN. Link quality can be highly affected by node positioning, and by external factors. A situation where this occurred led to reduced network connectivity during the first deployment. This situation was studied, and some conclusions that led to improvements were made. Node 13 was placed inside the pulpit at the floor that can be seen at the

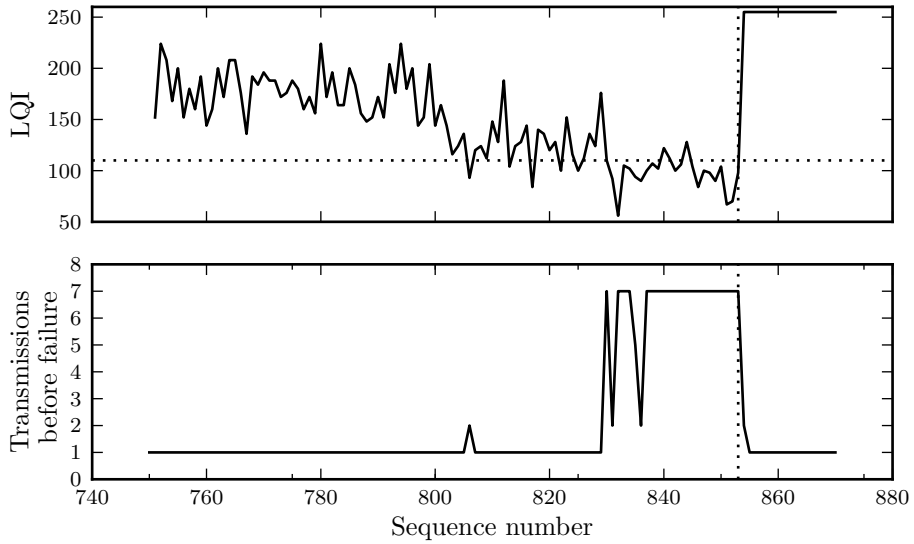


Figure 8.4: Link quality reduction between node 13 and its preferred parent

This figure shows retransmissions and LQI between node 13 and its preferred parent. The maximum amount of retransmissions are 6. A new preferred parent is selected at sequence number 853

right side of the picture in 8.2c. This pulpit was moved for some events. Radio communication between this node and other nodes could easily be a subject to obstructions caused surrounding people. Figure 13 shows a situation where this most likely happened. The preferred base station for node 13 was the base station (node 0). This link is working optimally with no retransmission, even though the LQI is not very high. After a while we can see the LQI dropping, and the number of retransmissions increasing. The nodes were configured with a maximum amount of 6 retransmission attempts before dropping a packet. This limit was reached by node 13 trying to reach the preferred parent many times. As the interval between sequence-numbers were 5 minutes, this situation did not change for over 2 hours, which is too slow for errors like this. After a while a global repair was triggered. Node 13 joined the new version at number 854. The base station was replaced as the preferred parent by node 10. This resolved the situation, and node 13 did not have to do any retransmissions to reach its preferred parent. Some conclusions were made from this situation, and some improvements implemented. Some of these conclusions were:

- Due to long intervals between DIO messages, the link information was not updated very often. The information in the neighbor table was therefore very old, and could not be used to safely find the routes. The global repair fixed this by resetting the trickle timer and updating the neighbor table.
- Since there was no interaction between the forwarding layer and RPL, there was no way that RPL could know that a bad route was chosen. RPL could therefore not fix a bad situation like this.

LQI	ETX
>220	1
>200	2
>180	3
>160	4
>140	5
>100	7
All else	+21

Table 8.1: LQI thresholds and link penalties in implementation 2

8.5.2 Improvements to the first implementation

In order to improve the results from the first deployment, some changes were made:

- Interaction between DataSend and RPL was introduced in order to keep the link information in the neighbor table more up to date. Message snooping was used in order to get this information from nodes that were not sending messages directly to this node
- A command was created that let DataSend signal RPL that the preferred parent was unavailable. This command decreased the metric associated with the preferred parent, and also marks it as unreachable. Both of these would ensure that the node would not be considered a parent in the next calculation, and also would have to regain its LQI before being chosen as a new route.
- Local repair should be enabled in order to let nodes choose new parents if no parents could be chosen within the given rank restrictions.

8.5.3 The second implementation

The second implementation introduced some of the changes discussed in the previous section. The nodes were deployed in the same locations, but reprogrammed with the new implementation. In order to try to increase network lifetime, the LPL sleeping interval was increased from 50 to 320ms. When using broadcast messages with LPL, the messages cannot be stopped transmitting upon reception of an acknowledgment. The minimum DIO interval was therefore increased to 512ms, in order to be larger than the sleep interval of the LPL.

For the second implementation, the link cost thresholds were changed to be more like the intended values described in chapter 5.2. In the first implementation, a human coding error caused all links with a LQI above 240 to get an ETX of 1, and all else to get an ETX of 2. In implementation 2 this threshold was reduced, and several more thresholds were introduced, which can be seen in table 8.1.

Problems during the second implementation

The second implementation showed some problems were many nodes disappeared simultaneously. This behavior will be described later, but was mostly caused by a bug where changes to the preferred parent node were not considered if the better route did not exceed the parent switch threshold. This bug also caused the rank

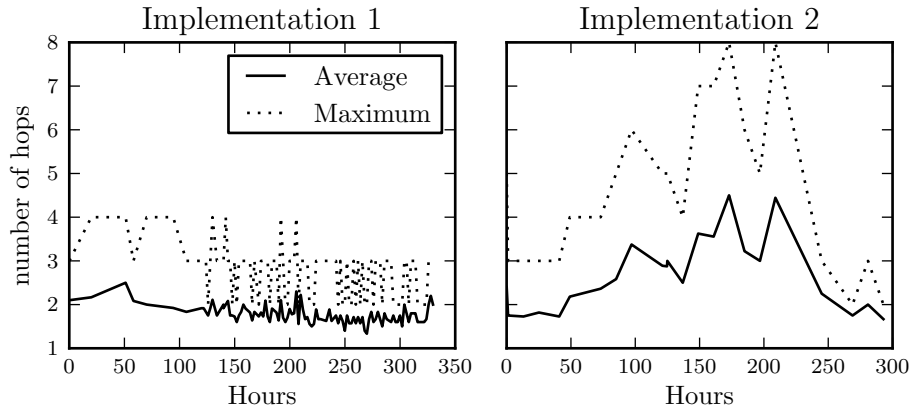


Figure 8.5: Hop number in RPL versions during the two implementations

of a node to not get affected by changes in the parents node, especially when the parent's node was increased.

8.6 Analysis

8.6.1 Hop count

Figure 8.5 shows the average and maximum number of hops for implementation 1 and 2. The average and maximum number of hops is pretty stable in implementation 1, where it averages around 2 hops, with a maximum at 3 or 4 hops. The situation for implementation 2 shows a very different routing topology. It starts at an average of about two hops, but then gradually increases. The maximum amount of hops is 8, and the maximum average is 4. This is a seems like an extremely high amount given that there are 10 active nodes at the time (see figure 8.6), and the first implementation showed much smaller hop counts. This was clearly not a good solution

The situation of a very large hop count seems to improve again at 220 hours, but looks can be deceiving. The hop count drop is caused by a drop in the active number of nodes, and these nodes had some of the longest routes. Figure 8.8 shows us that many of the disappeared nodes were within the energy requirement for operation (2.7 to 3.6V) [46], and low remaining energy was therefore most likely not the cause of the lost connectivity. It would also be weird if that many nodes lost battery at the same moment.

Figure 8.7 shows the routing topology right before many of the nodes disappears. We can see that many of the nodes that are close to root chose weird routes. Take for instance node 2, which is in the same room as the base station, but chooses the preferred parent of 9, making the route to the base-station 2->9->13->10-0. Many of the nodes that had the base station as the preferred parent in the first implementation (see fig 8.2c), are now taking very long routes. It can be seen that one of the nodes that disappears has low remaining battery voltage. Studying the data shows that this is node 11. Since this node works as a router for 4 nodes, it is likely that this node lost communication to its preferred parent (not

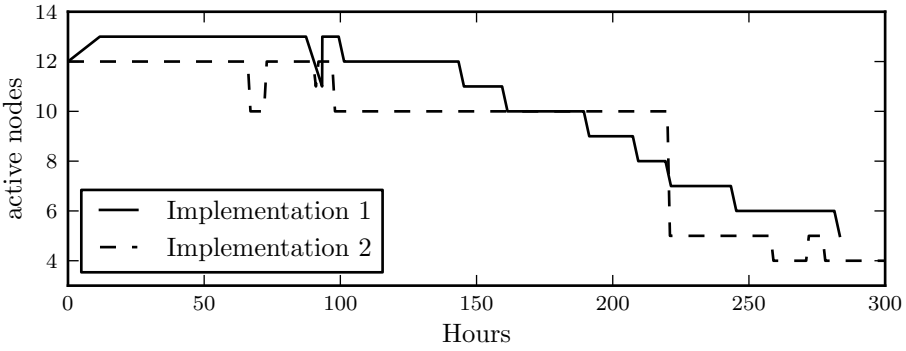


Figure 8.6: Number of active nodes during the two implementations

Note that there should be one less active node in the second implementation, as battery replacement was not possible for node 14 due to restricted access to the area.

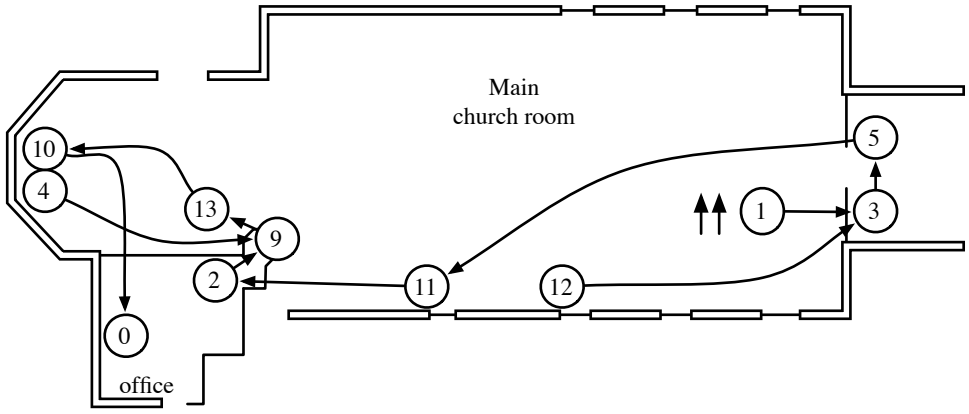


Figure 8.7: Weird routing topology

necessarily because of low battery), and brought all the other nodes with it. The implementation bugs that existed in this implementation could cause the nodes to not care about the increased rank for example used doing local repair, and thus not notice that the preferred parent became invalid. Another bug also caused the calculated rank associated with a parent node and its link to overload and wrap over to zero. The wrapping would cause an invalid parent to seem like a much better parent. If the path cost between a node and this parent were 1, the resulting rank would be 0, which would in turn be better than the DODAG root.

The major difference in the hop count between the first and second implementation is most likely caused by the more conservative values used in the LQI to ETX calculation. Additionally, it is likely that bugs made these long routes extra bad, as changes in rank to the preferred parent was ignored.

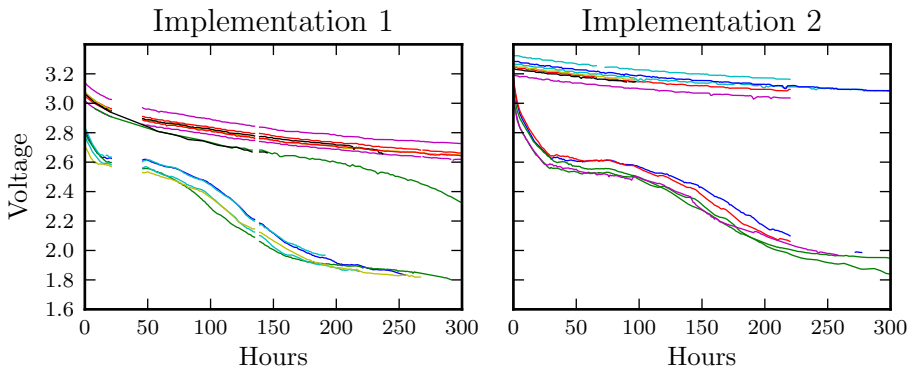


Figure 8.8: Battery voltage for all nodes during the two implementations

8.6.2 Battery

Figure 8.8 shows the battery voltage in all the nodes in the network. We can see that 5 nodes have less voltage than the remaining nodes. 5 out of the deployed nodes have a sensor board with GPS units equipped. It is not known which nodes this is, but there is a chance that these are the nodes with less voltage. The large initial drop in voltage seen almost instantly in these nodes causes this belief. If these are not the nodes with GPS units attached, these units are most likely forwarding many messages, or located in an area with much over hearing between neighbors.

8.6.3 Packet Reception Ratio

Although no numbers are represented, there has been a high Packet Reception Ratio. Numbers in the first implementation would be kinda useless, as the missing local repair would lead to a lot of packet loss during bad periods. The deployment has provided measurement for most of the expected 5-minute interval measurements, besides these.

Chapter 9

Conclusion

This thesis has implemented and deployed a WSN using TinyOS and RPL. The Minimum Rank with Hystersis Objective Function (MRHOF) is used with the ETX metric.

A test was done to find the correlation between link properties like Link quality indicator (LQI) and channel energy against Packet Reception Ratio (PRR). A large correlation between average LQI and PRR was found.

ETX is a widely used metric in WSN as it accounts for loss ratio and link asymmetry, and provides routes with a reasonable hop count. The ETX is usually calculated using information based on real communication, this information can be obtained using probes. These probes create network overhead, and thus increases radio energy consumption. This technique is therefore not very suitable for WSNs. Several RPL implementations use an approach where ETX is given an initial value, which is updated when the link is being actively used. This initial value can be either high, which can prevent the best parent from being tested and thus selected, or it can be low, leading to many switches between parents. This thesis proposes to use LQI to calculate the initial ETX values. The calculated ETX can provide a good guidance for the experienced ETX. Using the LQI does not increase network overhead, but still keeps value up to date for all nodes. TinyOS uses BoX-MAC 2 for Low Power Listening. This MAC layer detects channel activity using CCA, and keeps the radio awake for the next packet if activity is detected. Since the radio always reads a complete message, all packets can be used to update the LQI values without increasing power consumption. This enables us to easily keep the LQI up to date.

A WSN has been deployed where the ETX is directly calculated from LQI, without taking much account for experienced ETX. Results from this shows that the calculation from LQI to ETX is of great importance. A conservative approach for the LQI to ETX calculation led to routes with very large hop counts. Tests from a less conservative approach showed routes with shorter hop counts, but still with high delivery ratios. It is possible that some of these results are due to implementation errors that have later been resolved but not tested, this is left for future work.

The deployed network did serve its purpose of gathering temperature and humidity from the environment with a low loss ratio. The operating lifetime of the network was a bit disappointing. The Low Power Listening (LPL) sleeping

period, which controls the duty cycling of the radio, was increased in order to try to gain a longer lifetime. This did not give any clear results, as an implementation bug seems to have cause nodes to loose connectivity before they were out of power.

The implementation bugs are most likely fixed, but have not yet been tested in the real life deployment.

9.1 Future work

Some future work is proposed. The bug fixes in the implementation should first be checked. Further experience with the LQI to ETX calculation would also be beneficial. The results from the last implementation shows a situation where routes with a large hop count is used. The first implementation shows that shorter routes with good delivery ratios existed. The long routes in the second implementation are most likely a result of choosing a calculation from LQI to ETX that is too conservative. Less conservative values should be tested for this calculation.

The current implementation only accounts for the experienced ETX if a node is unreachable. The experienced ETX should be used to a greater extent. This could for example be done using a correction value for the LQI calculated from the actual ETX. This would make the LQI to ETX calculation work more like an initial value, which could be a great improvement over similar methods. This calculation is however very hardware dependent as the calculation of LQI is not standardized in any way.

Chapter 10

Bibliography

[1] ATMEL, "Atmel rf230 datasheet."

[2] I. Dietrich and F. Dressler, "On the lifetime of wireless sensor networks," *ACM Trans. Sen. Netw.*, vol. 5, pp. 5:1–5:39, Feb. 2009.

[3] C. Poellabauer and W. Dargie, *Fundamentals of Wireless Sensor Networks: Theory and Practice*. Wiley, 2011.

[4] "IEEE Standard for Local and metropolitan area networks–Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)," *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)*, pp. 1 –314, 5 2011.

[5] Crossbow, "Crossbow iris datasheet."

[6] J. Al-Karaki and A. Kamal, "Routing techniques in wireless sensor networks: a survey," *Wireless Communications, IEEE*, vol. 11, pp. 6 – 28, dec. 2004.

[7] J. Al-Karaki, R. Ul-Mustafa, and A. Kamal, "Data aggregation in wireless sensor networks - exact and approximate algorithms," in *High Performance Switching and Routing, 2004. HPSR. 2004 Workshop on*, pp. 241 – 245, 2004.

[8] IETF, "Charter for "Routing Over Low power and Lossy networks" (roll) WG," February 2008.

[9] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks." RFC 6550 (Proposed Standard), Mar. 2012.

[10] J. Hui, J. Vasseur, D. Culler, and V. Manral, "An IPv6 Routing Header for Source Routes with the Routing Protocol for Low-Power and Lossy Networks (RPL)." RFC 6554 (Proposed Standard), Mar. 2012.

[11] P. Levis, T. Clausen, J. Hui, O. Gnawali, and J. Ko, "The Trickle Algorithm." RFC 6206 (Proposed Standard), Mar. 2011.

[12] J. Vasseur, M. Kim, K. Pister, N. Dejean, and D. Barthel, "Routing Metrics Used for Path Calculation in Low-Power and Lossy Networks." RFC 6551 (Proposed Standard), Mar. 2012.

[13] D. S. J. De Couto, D. Aguayo, J. Bicket, and R. Morris, "A high-throughput path metric for multi-hop wireless routing," in *Proceedings of the 9th ACM International Conference on Mobile Computing and Networking (MobiCom '03)*, (San Diego, California), September 2003.

[14] S. Dawans, S. Duquennoy, and O. Bonaventure, "On link estimation in dense rpl deployments," in *Proceedings of the International Workshop on Practical Issues in Building Sensor Network Applications (IEEE SenseApp 2012)*, (Florida, USA), October 2012.

[15] P. Thubert, "Objective Function Zero for the Routing Protocol for Low-Power and Lossy Networks (RPL)." RFC 6552 (Proposed Standard), Mar. 2012.

[16] O. Gnawali and P. Levis, "The Minimum Rank with Hysteresis Objective Function." RFC 6719 (Proposed Standard), Sept. 2012.

[17] J. W. Hui and D. E. Culler, "Ip is dead, long live ip for wireless sensor networks," in *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, (New York, NY, USA), pp. 15–28, ACM, 2008.

[18] Crossbow, "Crossbow mib510 iris datasheet."

[19] Crossbow, "Crossbow mts420 sensorboard datasheet."

[20] Sensirion, *Sensirion SHT1x DataSheet.* Sensirion.

[21] N. Baccour, A. Koubâa, L. Mottola, M. A. Zúñiga, H. Youssef, C. A. Boano, and M. Alves, "Radio link quality estimation in wireless sensor networks: A survey," *ACM Trans. Sen. Netw.*, vol. 8, pp. 34:1–34:33, Sept. 2012.

[22] K. Srinivasan and P. Levis, "Rssi is under appreciated," in *In Proceedings of the Third Workshop on Embedded Networked Sensors (EmNets*, 2006.

[23] "Tinyos webpage."

[24] P. Levis, "Experiences from a decade of tinyos development," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 207–220, USENIX Association, 2012.

[25] P. Levis, *TinyOS Programming*, July 2009.

[26] "Getting started with tinyos," sept. 2012.

[27] T. wiki, "Modules and the tinyos exectuion model," May 2010.

[28] V. Handziski, J. Polastre, J.-H. Hauer, C. Sharp, A. Wolisz, D. Culler, and D. Gay, "Hardware abstraction architecture." TinyOS Enhancement Proposal 2.

[29] J. Hui, P. Levis, and D. Moss, "Tinyos 802.15.4 frames." TinyOS Enhancement Proposal 125, June 2008.

[30] P. Levis, "Packet protocols." TinyOS Enhancement Proposal 116.

[31] "Tinyos receive.nc implementation."

[32] D. Gay, P. Levis, D. Culler, and E. Brewer, *nesC 1.3 Language Reference*, July 2009.

[33] P. Levis, "message_t." TinyOS Enhancement Proposal 111.

[34] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification." RFC 2460 (Draft Standard), Dec. 1998. Updated by RFCs 5095, 5722, 5871, 6437, 6564.

[35] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks." RFC 4944 (Proposed Standard), Sept. 2007. Updated by RFCs 6282, 6775.

[36] J. Ko, J. Eriksson, N. Tsiftes, S. Dawson-haggerty, A. Terzis, A. Dunkels, and D. Culler, "Contikirpl and tinyrpl: Happy together," in *In Proceedings of the workshop on Extending the Internet to Low power and Lossy Networks (IP+SN 2011*, 2011.

[37] T. wiki, "Tinyos: Writing low power applications."

[38] R. Szewczyk, P. Levis, M. Turon, L. Nachman, P. Buonadonna, and V. Handziski, "Microcontroller power management." TinyOS Enhancement Proposal 112, January 2007.

[39] D. Moss, J. Hui, and K. Klues, "Low power listening." TinyOS Enhancement Proposal 105.

[40] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, (New York, NY, USA), pp. 95–107, ACM, 2004.

[41] M. Buettner, G. V. Yee, E. Anderson, and R. Han, "X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, (New York, NY, USA), pp. 307–320, ACM, 2006.

[42] D. Moss and P. Levis, "Box-macs: Exploiting physical and link layer boundaries in low-power networking," tech. rep., Stanford University, 2008.

[43] D. Moss, J. Hui, P. Levis, and J. I. Choi, "Low power listening." TinyOS Enhancement Proposal 126, June 2007.

[44] R. Elz and R. Bush, "Serial Number Arithmetic." RFC 1982 (Proposed Standard), Aug. 1996.

[45] R. J. Perlman, "Fault-tolerant broadcast of routing information," in *Computer Networks 7*, pp. 395–405, 1983.

[46] Crossbow, "Crossbow iris hardware reference."